

```

Field field = modifiedClass.getDeclaredField("myField1"); // Stores the
// field from
5 // the modified
// class.

// In this example, the field is a byte field.
while (DRT.isRunning){

10 ms.receive(dp); // Receive the previously sent buffer from the network.

    byte[] b = dp.getData();

    if (b[0] == nameTag){ // Check the nametags match.
15 field.setByte(null, b[1]); // Write the value from the network packet
// into the field location in memory.

    }

20 }
// END

```

- A5. The fifth excerpt is an disassembled compiled form of the example.java application of Annexure A7, which performs a memory manipulation operation (putstatic and putfield).

```

Method void setValues(int, int)
0 iload_1
1 putstatic #3 <Field int staticValue>
30 4 aload_0
5 iload_2
6 putfield #2 <Field int instanceValue>
9 return

```

- A6. The sixth excerpt is the disassembled compiled form of the same example application in Annexure A5 after modification has been performed by FieldLoader.java of Annexure A11, in accordance with Fig. 9 of this invention. The modifications are highlighted in bold.

```

40 Method void setValues(int, int)
    0 iload_1
    1 putstatic #3 <Field int staticValue>
    4 ldc #4 <String "example">
    6 iconst_0
45 7 invokestatic #5 <Method void alert(java.lang.Object, int)>
    10 aload_0
    11 iload_2
    12 putfield #2 <Field int instanceValue>
    15 aload_0
50 16 iconst_1
    17 invokestatic #5 <Method void alert(java.lang.Object, int)>
    20 return

```

A7. The seventh excerpt is the source-code of the example.java application used in excerpt A5 and A6. This example application has two memory locations (staticValue and instanceValue) and performs two memory manipulation operations.

```

5  import java.lang.*;

   public class example{

10     /** Shared static field. */
       public static int staticValue = 0;

       /** Shared instance field. */
       public int instanceValue = 0;

15     /** Example method that writes to memory (instance field). */
       public void setValues(int a, int b){

           staticValue = a;

20           instanceValue = b;

       }

25 }

```

A8. The eighth excerpt is the source-code of FieldAlert.java which corresponds to step 125 and arrow 127 of Fig. 12, and which requests a thread 121/1 executing FieldSend.java of the "distributed run-time" 71 to propagate a changed value and identity pair to the other machines M1...Mn.

```

30  import java.lang.*;
   import java.util.*;
   import java.net.*;
   import java.io.*;
   public class FieldAlert{

40     /** Table of alerts. */
       public final static Hashtable alerts = new Hashtable();

       /** Object handle. */
       public Object reference = null;

45     /** Table of field alerts for this object. */
       public boolean[] fieldAlerts = null;

       /** Constructor. */
       public FieldAlert(Object o, int initialFieldCount){
           reference = o;
           fieldAlerts = new boolean[initialFieldCount];
       }

55     /** Called when an application modifies a value. (Both objects and
       classes) */
       public static void alert(Object o, int fieldID){

           // Lock the alerts table.
           synchronized (alerts){

60               FieldAlert alert = (FieldAlert) alerts.get(o);

               if (alert == null){ // This object hasn't been alerted already,

```

```

// so add to alerts table.
alert = new FieldAlert(o, fieldID + 1);
alerts.put(o, alert);
}
5
if (fieldID >= alert.fieldAlerts.length){
// Ok, enlarge fieldAlerts array.
boolean[] b = new boolean[fieldID+1];
10
System.arraycopy(alert.fieldAlerts, 0, b, 0,
alert.fieldAlerts.length);
alert.fieldAlerts = b;
}

// Record the alert.
15
alert.fieldAlerts[fieldID] = true;

// Mark as pending.
FieldSend.pending = true; // Signal that there is one or more
// propagations waiting.
20

// Finally, notify the waiting FieldSend thread(s)
if (FieldSend.waiting){
FieldSend.waiting = false;
25
alerts.notify();
}
}
30
}
}

```

35 A9. The ninth excerpt is the source-code of FieldSend.java which corresponds to step 128 of Fig. 12, and waits for a request/notification generated by FieldAlert.java of A8 corresponding to step 125 and arrow 127, and which propagates a changed value/identity pair requested of it by FieldAlert.java, via network 53.

```

import java.lang.*;
import java.lang.reflect.*;
import java.util.*;
import java.net.*;
import java.io.*;
45
public class FieldSend implements Runnable{

/** Protocol specific values. */
public final static int CLOSE = -1;
50
public final static int NACK = 0;
public final static int ACK = 1;
public final static int PROPAGATE_OBJECT = 10;
public final static int PROPAGATE_CLASS = 20;

/** FieldAlert network values. */
public final static String group =
System.getProperty("FieldAlert_network_group");
public final static int port =
Integer.parseInt(System.getProperty("FieldAlert_network_port"));
60

/** Table of global ID's for local objects. (hashcode-to-globalID
mappings) */
public final static Hashtable objectToGlobalID = new Hashtable();

65
/** Table of global ID's for local classnames. (classname-to-globalID
mappings) */

```

```

public final static Hashtable classNameToGlobalID = new Hashtable();

/** Pending. True if a propagation is pending. */
public static boolean pending = false;

5  /** Waiting. True if the FieldSend thread(s) are waiting. */
    public static boolean waiting = false;

10 /** Background send thread. Propagates values as this thread is alerted
    to their alteration. */
    public void run(){

        System.out.println("FieldAlert_network_group=" + group);
        System.out.println("FieldAlert_network_port=" + port);

15    try{

        // Create a DatagramSocket to send propagated field values.
        DatagramSocket datagramSocket =
20        new DatagramSocket(port, InetAddress.getByName(group));

        // Next, create the buffer and packet for all transmissions.
        byte[] buffer = new byte[512];           // Working limit of 512 bytes
                                                // per packet.
25        DatagramPacket datagramPacket =
            new DatagramPacket(buffer, 0, buffer.length);

        while (!Thread.interrupted()){

30            Object[] entries = null;

            // Lock the alerts table.
            synchronized (FieldAlert.alerts){

35                // Await for an alert to propagate something.
                while (!pending){
                    waiting = true;
                    FieldAlert.alerts.wait();
                    waiting = false;
40                }

                pending = false;

                entries = FieldAlert.alerts.entrySet().toArray();

45                // Clear alerts once we have copied them.
                FieldAlert.alerts.clear();

            }

50            // Process each object alert in turn.
            for (int i=0; i<entries.length; i++){

                FieldAlert alert = (FieldAlert) entries[i];

55                int index = 0;
                datagramPacket.setLength(buffer.length);

                Object reference = null;
                if (alert.reference instanceof String){
                    // PROPAGATE_CLASS field operation.

60                    buffer[index++] = (byte) ((PROPAGATE_CLASS >> 24) & 0xff);
                    buffer[index++] = (byte) ((PROPAGATE_CLASS >> 16) & 0xff);
                    buffer[index++] = (byte) ((PROPAGATE_CLASS >> 8) & 0xff);
                    buffer[index++] = (byte) ((PROPAGATE_CLASS >> 0) & 0xff);
65

```

```

String name = (String) alert.reference;
int length = name.length();
buffer[index++] = (byte) ((length >> 24) & 0xff);
buffer[index++] = (byte) ((length >> 16) & 0xff);
5   buffer[index++] = (byte) ((length >> 8) & 0xff);
    buffer[index++] = (byte) ((length >> 0) & 0xff);

byte[] bytes = name.getBytes();
10   System.arraycopy(bytes, 0, buffer, index, length);
    index += length;

} else {                                // PROPAGATE_OBJECT field operation.

    buffer[index++] =
15   (byte) ((PROPAGATE_OBJECT >> 24) & 0xff);
    buffer[index++] =
        (byte) ((PROPAGATE_OBJECT >> 16) & 0xff);
    buffer[index++] = (byte) ((PROPAGATE_OBJECT >> 8) & 0xff);
    buffer[index++] = (byte) ((PROPAGATE_OBJECT >> 0) & 0xff);

    int globalID = ((Integer)
20   objectToGlobalID.get(alert.reference)).intValue();

    buffer[index++] = (byte) ((globalID >> 24) & 0xff);
    buffer[index++] = (byte) ((globalID >> 16) & 0xff);
25   buffer[index++] = (byte) ((globalID >> 8) & 0xff);
    buffer[index++] = (byte) ((globalID >> 0) & 0xff);

    reference = alert.reference;

30   }

    // Use reflection to get a table of fields that correspond to
    // the field indexes used internally.
    Field[] fields = null;
    if (reference == null) {
        fields = FieldLoader.loadClass((String)
35   alert.reference).getDeclaredFields();
    } else {
        fields = alert.reference.getClass().getDeclaredFields();
    }

    // Now encode in batch mode the fieldID/value pairs.
    for (int j=0; j<alert.fieldAlerts.length; j++){
45   if (alert.fieldAlerts[j] == false)
        continue;

    buffer[index++] = (byte) ((j >> 24) & 0xff);
    buffer[index++] = (byte) ((j >> 16) & 0xff);
50   buffer[index++] = (byte) ((j >> 8) & 0xff);
    buffer[index++] = (byte) ((j >> 0) & 0xff);

    // Encode value.
    Class type = fields[j].getType();
    if (type == Boolean.TYPE) {
        buffer[index++] = (byte)
55   (fields[j].getBoolean(reference)? 1 : 0);
    } else if (type == Byte.TYPE) {
        buffer[index++] = fields[j].getByte(reference);
    } else if (type == Short.TYPE) {
        short v = fields[j].getShort(reference);
        buffer[index++] = (byte) ((v >> 8) & 0xff);
        buffer[index++] = (byte) ((v >> 0) & 0xff);
60   } else if (type == Character.TYPE) {
        char v = fields[j].getChar(reference);
        buffer[index++] = (byte) ((v >> 8) & 0xff);
65   }

```

```

        buffer[index++] = (byte) ((v >> 0) & 0xff);
    } else if (type == Integer.TYPE) {
        int v = fields[j].getInt(reference);
        buffer[index++] = (byte) ((v >> 24) & 0xff);
        buffer[index++] = (byte) ((v >> 16) & 0xff);
        buffer[index++] = (byte) ((v >> 8) & 0xff);
        buffer[index++] = (byte) ((v >> 0) & 0xff);
    } else if (type == Float.TYPE) {
        int v = Float.floatToIntBits(
            fields[j].getFloat(reference));
        buffer[index++] = (byte) ((v >> 24) & 0xff);
        buffer[index++] = (byte) ((v >> 16) & 0xff);
        buffer[index++] = (byte) ((v >> 8) & 0xff);
        buffer[index++] = (byte) ((v >> 0) & 0xff);
    } else if (type == Long.TYPE) {
        long v = fields[j].getLong(reference);
        buffer[index++] = (byte) ((v >> 56) & 0xff);
        buffer[index++] = (byte) ((v >> 48) & 0xff);
        buffer[index++] = (byte) ((v >> 40) & 0xff);
        buffer[index++] = (byte) ((v >> 32) & 0xff);
        buffer[index++] = (byte) ((v >> 24) & 0xff);
        buffer[index++] = (byte) ((v >> 16) & 0xff);
        buffer[index++] = (byte) ((v >> 8) & 0xff);
        buffer[index++] = (byte) ((v >> 0) & 0xff);
    } else if (type == Double.TYPE) {
        long v = Double.doubleToLongBits(
            fields[j].getDouble(reference));
        buffer[index++] = (byte) ((v >> 56) & 0xff);
        buffer[index++] = (byte) ((v >> 48) & 0xff);
        buffer[index++] = (byte) ((v >> 40) & 0xff);
        buffer[index++] = (byte) ((v >> 32) & 0xff);
        buffer[index++] = (byte) ((v >> 24) & 0xff);
        buffer[index++] = (byte) ((v >> 16) & 0xff);
        buffer[index++] = (byte) ((v >> 8) & 0xff);
        buffer[index++] = (byte) ((v >> 0) & 0xff);
    } else {
        throw new AssertionError("Unsupported type.");
    }
}

// Now set the length of the datagram packet.
datagramPacket.setLength(index);

// Now send the packet.
datagramSocket.send(datagramPacket);

}

} catch (Exception e) {
    throw new AssertionError("Exception: " + e.toString());
}

}

}

```

A10. The tenth excerpt is the source-code of FieldReceive.java, which corresponds to steps 135 and 136 of Fig. 13, and which receives a propagated changed value and identity pair sent to it over the network 53 via FieldSend.java of annexure A9.

```
import java.lang.*;
```

```

import java.lang.reflect.*;
import java.util.*;
import java.net.*;
import java.io.*;

5 public class FieldReceive implements Runnable{

    /** Protocol specific values. */
    public final static int CLOSE = -1;
    10 public final static int NACK = 0;
    public final static int ACK = 1;
    public final static int PROPAGATE_OBJECT = 10;
    public final static int PROPAGATE_CLASS = 20;

    15 /** FieldAlert network values. */
    public final static String group =
        System.getProperty("FieldAlert_network_group");
    public final static int port =
        Integer.parseInt(System.getProperty("FieldAlert_network_port"));

    20 /** Table of global ID's for local objects. (globalID-to-hashcode
        mappings) */
    public final static Hashtable globalIDToObject = new Hashtable();

    25 /** Table of global ID's for local classnames. (globalID-to-classname
        mappings) */
    public final static Hashtable globalIDToClassName = new Hashtable();

    30 /** Called when an application is to acquire a lock. */
    public void run(){

        System.out.println("FieldAlert_network_group=" + group);
        System.out.println("FieldAlert_network_port=" + port);

    35 try{

        // Create a DatagramSocket to send propagated field values from
        MulticastSocket multicastSocket = new MulticastSocket(port);
        multicastSocket.joinGroup(InetAddress.getByAddress(group));

    40 // Next, create the buffer and packet for all transmissions.
        byte[] buffer = new byte[512]; // Working limit of 512
        // bytes per packet.

        DatagramPacket datagramPacket =
    45 new DatagramPacket(buffer, 0, buffer.length);

        while (!Thread.interrupted()){

            // Make sure to reset length.
            datagramPacket.setLength(buffer.length);

    50 // Receive the next available packet.
            multicastSocket.receive(datagramPacket);

            55 int index = 0, length = datagramPacket.getLength();

            // Decode the command.
            int command = (int) (((buffer[index++] & 0xff) << 24)
                | ((buffer[index++] & 0xff) << 16)
                | ((buffer[index++] & 0xff) << 8)
                | (buffer[index++] & 0xff));

            60 if (command == PROPAGATE_OBJECT){ // Propagate operation for
                // object fields.

                // Decode global id.
                int globalID = (int) (((buffer[index++] & 0xff) << 24)

```

```

        | ((buffer[index++] & 0xff) << 16)
        | ((buffer[index++] & 0xff) << 8)
        | (buffer[index++] & 0xff));

5      // Now, need to resolve the object in question.
      Object reference = globalIDToObject.get(
        new Integer(globalID));

10     // Next, get the array of fields for this object.
      Field[] fields = reference.getClass().getDeclaredFields();

      while (index < length){

        // Decode the field id.
15      int fieldID = (int) (((buffer[index++] & 0xff) << 24)
        | ((buffer[index++] & 0xff) << 16)
        | ((buffer[index++] & 0xff) << 8)
        | (buffer[index++] & 0xff));

20      // Determine value length based on corresponding field
      // type.
      Field field = fields[fieldID];
      Class type = field.getType();
      if (type == Boolean.TYPE){
25      boolean v = (buffer[index++] == 1 ? true : false);
        field.setBoolean(reference, v);
      }else if (type == Byte.TYPE){
        byte v = buffer[index++];
        field.setByte(reference, v);
30      }else if (type == Short.TYPE){
        short v = (short) (((buffer[index++] & 0xff) << 8)
        | (buffer[index++] & 0xff));
        field.setShort(reference, v);
      }else if (type == Character.TYPE){
35      char v = (char) (((buffer[index++] & 0xff) << 8)
        | (buffer[index++] & 0xff));
        field.setChar(reference, v);
      }else if (type == Integer.TYPE){
        int v = (int) (((buffer[index++] & 0xff) << 24)
40      | ((buffer[index++] & 0xff) << 16)
        | ((buffer[index++] & 0xff) << 8)
        | (buffer[index++] & 0xff));
        field.setInt(reference, v);
      }else if (type == Float.TYPE){
45      int v = (int) (((buffer[index++] & 0xff) << 24)
        | ((buffer[index++] & 0xff) << 16)
        | ((buffer[index++] & 0xff) << 8)
        | (buffer[index++] & 0xff));
        field.setFloat(reference, Float.intBitsToFloat(v));
50      }else if (type == Long.TYPE){
        long v = (long) (((buffer[index++] & 0xff) << 56)
        | ((buffer[index++] & 0xff) << 48)
        | ((buffer[index++] & 0xff) << 40)
        | ((buffer[index++] & 0xff) << 32)
55      | ((buffer[index++] & 0xff) << 24)
        | ((buffer[index++] & 0xff) << 16)
        | ((buffer[index++] & 0xff) << 8)
        | (buffer[index++] & 0xff));
        field.setLong(reference, v);
60      }else if (type == Double.TYPE){
        long v = (long) (((buffer[index++] & 0xff) << 56)
        | ((buffer[index++] & 0xff) << 48)
        | ((buffer[index++] & 0xff) << 40)
        | ((buffer[index++] & 0xff) << 32)
65      | ((buffer[index++] & 0xff) << 24)
        | ((buffer[index++] & 0xff) << 16)
        | ((buffer[index++] & 0xff) << 8)
        | (buffer[index++] & 0xff) << 8)

```

```

        | (buffer[index++] & 0xff));
        field.setDouble(reference, Double.longBitsToDouble(v));
    }else{
        throw new AssertionError("Unsupported type.");
    }
}

}

else if (command == PROPAGATE_CLASS){ // Propagate an update
    // to class fields.

    // Decode the classname.
    int nameLength = (int) (((buffer[index++] & 0xff) << 24)
        | ((buffer[index++] & 0xff) << 16)
        | ((buffer[index++] & 0xff) << 8)
        | (buffer[index++] & 0xff));
    String name = new String(buffer, index, nameLength);
    index += nameLength;

    // Next, get the array of fields for this class.
    Field[] fields =
        FieldLoader.loadClass(name).getDeclaredFields();

    // Decode all batched fields included in this propagation
    // packet.
    while (index < length){
        // Decode the field id.
        int fieldID = (int) (((buffer[index++] & 0xff) << 24)
            | ((buffer[index++] & 0xff) << 16)
            | ((buffer[index++] & 0xff) << 8)
            | (buffer[index++] & 0xff));

        // Determine field type to determine value length.
        Field field = fields[fieldID];
        Class type = field.getType();
        if (type == Boolean.TYPE){
            boolean v = (buffer[index++] == 1 ? true : false);
            field.setBoolean(null, v);
        }else if (type == Byte.TYPE){
            byte v = buffer[index++];
            field.setByte(null, v);
        }else if (type == Short.TYPE){
            short v = (short) (((buffer[index++] & 0xff) << 8)
                | (buffer[index++] & 0xff));
            field.setShort(null, v);
        }else if (type == Character.TYPE){
            char v = (char) (((buffer[index++] & 0xff) << 8)
                | (buffer[index++] & 0xff));
            field.setChar(null, v);
        }else if (type == Integer.TYPE){
            int v = (int) (((buffer[index++] & 0xff) << 24)
                | ((buffer[index++] & 0xff) << 16)
                | ((buffer[index++] & 0xff) << 8)
                | (buffer[index++] & 0xff));
            field.setInt(null, v);
        }else if (type == Float.TYPE){
            int v = (int) (((buffer[index++] & 0xff) << 24)
                | ((buffer[index++] & 0xff) << 16)
                | ((buffer[index++] & 0xff) << 8)
                | (buffer[index++] & 0xff));
            field.setFloat(null, Float.intBitsToFloat(v));
        }else if (type == Long.TYPE){
            long v = (long) (((buffer[index++] & 0xff) << 56)
                | ((buffer[index++] & 0xff) << 48)
                | ((buffer[index++] & 0xff) << 40)
                | ((buffer[index++] & 0xff) << 32)

```

```

        | ((buffer[index++] & 0xff) << 24)
        | ((buffer[index++] & 0xff) << 16)
        | ((buffer[index++] & 0xff) << 8)
        | (buffer[index++] & 0xff));
5         field.setLong(null, v);
    }else if (type == Double.TYPE){
        long v = (long) (((buffer[index++] & 0xff) << 56)
        | ((buffer[index++] & 0xff) << 48)
        | ((buffer[index++] & 0xff) << 40)
10         | ((buffer[index++] & 0xff) << 32)
        | ((buffer[index++] & 0xff) << 24)
        | ((buffer[index++] & 0xff) << 16)
        | ((buffer[index++] & 0xff) << 8)
        | (buffer[index++] & 0xff));
15         field.setDouble(null, Double.longBitsToDouble(v));
    }else{
        // Unsupported field type.
        throw new AssertionError("Unsupported type.");
    }
}

    }

}

}
25 }catch (Exception e){
    throw new AssertionError("Exception: " + e.toString());
}
}
30 }

```

A11. FieldLoader.java

This excerpt is the source-code of FieldLoader.java, which modifies an application program code, such as the example.java application code of Annexure A7, as it is being loaded into a JAVA virtual machine in accordance with steps 90, 91, 92, 103, and 94 of Fig. 10. FieldLoader.java makes use of the convenience classes of Annexures A12 through to A36 during the modification of a compiled JAVA classfile.

```
import java.lang.*;
import java.io.*;
45 import java.net.*;

public class FieldLoader extends URLClassLoader{

    public FieldLoader(URL[] urls){
50         super(urls);
    }

    protected Class findClass(String name)
    throws ClassNotFoundException{
55         ClassFile cf = null;

        try{

60             BufferedInputStream in =
```

```

        new BufferedInputStream(findResource(
            name.replace('.', '/').concat(".class")).openStream());

5         cf = new ClassFile(in);

    } catch (Exception e) { throw new ClassNotFoundException(e.toString()); }

    // Class-wide pointers to the ldc and alert index.
    int ldcindex = -1;
10    int alertindex = -1;

    for (int i=0; i<cf.methods_count; i++){
        for (int j=0; j<cf.methods[i].attributes_count; j++){
            if (!(cf.methods[i].attributes[j] instanceof Code_attribute))
15                continue;

            Code_attribute ca = (Code_attribute)
                cf.methods[i].attributes[j];

20            boolean changed = false;

            for (int z=0; z<ca.code.length; z++){
                if ((ca.code[z][0] & 0xff) == 179) { // Opcode for a
PUTSTATIC                                     // instruction.
25                changed = true;

                // The code below only supports fields in this class.
                // Thus, first off, check that this field is local to this
                // class.
                CONSTANT_Fieldref_info fi = (CONSTANT_Fieldref_info)
                    cf.constant_pool[(int) ((ca.code[z][1] & 0xff) << 8) |
                    (ca.code[z][2] & 0xff)];
                CONSTANT_Class_info ci = (CONSTANT_Class_info)
35                cf.constant_pool[fi.class_index];

                String className =
                    cf.constant_pool[ci.name_index].toString();
                if (!name.equals(className)){
                    throw new AssertionError("This code only supports
40                fields "
                        + "local to this class");
                }

                // Ok, now search for the fields name and index.
                int index = 0;
                CONSTANT_NameAndType_info ni = (CONSTANT_NameAndType_info)
                    cf.constant_pool[fi.name_and_type_index];
                String fieldName =
                    cf.constant_pool[ni.name_index].toString();
50                for (int a=0; a<cf.fields_count; a++){
                    String fn = cf.constant_pool[
                        cf.fields[a].name_index].toString();
                    if (fieldName.equals(fn)){
                        index = a;
                        break;
55                    }
                }

                // Next, realign the code array, making room for the
                // insertions.
                byte[] code2 = new byte[ca.code.length+3];
                System.arraycopy(ca.code, 0, code2, 0, z+1);
                System.arraycopy(ca.code, z+1, code2, z+4,
65                ca.code.length-(z+1));
                ca.code = code2;

```

```

// Next, insert the LDC_W instruction.
if (ldcindex == -1){
    CONSTANT_String_info csi =
        new CONSTANT_String_info(c1.name_index);
5   cp_info[] cpi = new cp_info(cf.constant_pool.length+1);
    System.arraycopy(cf.constant_pool, 0, cpi, 0,
        cf.constant_pool.length);
    cpi[cpi.length - 1] = csi;
    ldcindex = cpi.length-1;
10   cf.constant_pool = cpi;
    cf.constant_pool_count++;
}
ca.code[z+1] = new byte[3];
ca.code[z+1][0] = (byte) 19;
15   ca.code[z+1][1] = (byte) ((ldcindex >> 8) & 0xff);
    ca.code[z+1][2] = (byte) (ldcindex & 0xff);

// Next, insert the SIPUSH instruction.
ca.code[z+2] = new byte[3];
ca.code[z+2][0] = (byte) 17;
ca.code[z+2][1] = (byte) ((index >> 8) & 0xff);
ca.code[z+2][2] = (byte) (index & 0xff);

20   // Finally, insert the INVOKESTATIC instruction.
    if (alertindex == -1){
        // This is the first time this class is encountering
the
        // alert instruction, so have to add it to the constant
        // pool.
30   cp_info[] cpi = new cp_info(cf.constant_pool.length+6);
        System.arraycopy(cf.constant_pool, 0, cpi, 0,
            cf.constant_pool.length);
        cf.constant_pool = cpi;
35   cf.constant_pool_count += 6;

        CONSTANT_Utf8_info u1 =
            new CONSTANT_Utf8_info("FieldAlert");
        cf.constant_pool[cf.constant_pool.length-6] = u1;
40   CONSTANT_Class_info c1 = new CONSTANT_Class_info(
            cf.constant_pool_count-6);
        cf.constant_pool[cf.constant_pool.length-5] = c1;

        u1 = new CONSTANT_Utf8_info("alert");
        cf.constant_pool[cf.constant_pool.length-4] = u1;
45   u1 = new CONSTANT_Utf8_info("(Ljava/lang/Object;)V");
        cf.constant_pool[cf.constant_pool.length-3] = u1;

50   CONSTANT_NameAndType_info n1 =
        new CONSTANT_NameAndType_info(
            cf.constant_pool.length-4, cf.constant_pool.length-
3);
        cf.constant_pool[cf.constant_pool.length-2] = n1;

        CONSTANT_Methodref_info m1 = new
CONSTANT_Methodref_info(
60   cf.constant_pool.length-5, cf.constant_pool.length-
2);
        cf.constant_pool[cf.constant_pool.length-1] = m1;
        alertindex = cf.constant_pool.length-1;
    }
    ca.code[z+3] = new byte[3];
    ca.code[z+3][0] = (byte) 184;
    ca.code[z+3][1] = (byte) ((alertindex >> 8) & 0xff);
    ca.code[z+3][2] = (byte) (alertindex & 0xff);
65

```

```

// And lastly, increase the CODE_LENGTH and
ATTRIBUTE_LENGTH // values.
5          ca.code_length += 9;
          ca.attribute_length += 9;
          }
10      }
          // If we changed this method, then increase the stack size by
one.
15      if (changed){
          ca.max_stack++;          // Just to make sure.
      }
      }
20  }

try(
25      ByteArrayOutputStream out = new ByteArrayOutputStream();
      cf.serialize(out);
      byte[] b = out.toByteArray();
30      return defineClass(name, b, 0, b.length);
    )catch (Exception e){
        throw new ClassNotFoundException(name);
    }
35  }
}

```

#### 40 A12. Attribute\_info.java

Convenience class for representing attribute\_info structures within ClassFiles.

```

import java.lang.*;
import java.io.*;

45 /** This abstract class represents all types of attribute_info
 * that are used in the JVM specifications.
 *
 * All new attribute_info subclasses are to always inherit from this
 * class.
 */
50 public abstract class attribute_info{

    public int attribute_name_index;
    public int attribute_length;

55 /** This is used by subclasses to register themselves
 * to their parent classFile.
 */
    attribute_info(ClassFile cf){}

60 /** Used during input serialization by ClassFile only. */
    attribute_info(ClassFile cf, DataInputStream in)
        throws IOException{
        attribute_name_index = in.readInt();
        attribute_length = in.readInt();
65    }
}

```

```

    /** Used during output serialization by ClassFile only. */
    void serialize(DataOutputStream out)
    throws IOException{
5      out.writeChar(attribute_name_index);
      out.writeInt(attribute_length);
    }

    /** This class represents an unknown attribute_info that
     * this current version of classfile specification does
     * not understand.
     */
    public final static class Unknown extends attribute_info{
15      byte[] info;

      /** Used during input serialization by ClassFile only. */
      Unknown(ClassFile cf, DataInputStream in)
      throws IOException{
20        super(cf, in);
        info = new byte[attribute_length];
        in.read(info, 0, attribute_length);
      }

      /** Used during output serialization by ClassFile only. */
      void serialize(DataOutputStream out)
      throws IOException{
25        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        super.serialize(out);
        out.write(info, 0, attribute_length);
30      }
    }
35 }

```

### A13. ClassFile.java

Convenience class for representing ClassFile structures.

```

40 import java.lang.*;
    import java.io.*;
    import java.util.*;

    /** The ClassFile follows verbatim from the JVM specification. */
45 public final class ClassFile {

    public int magic;
    public int minor_version;
    public int major_version;
50 public int constant_pool_count;
    public cp_info[] constant_pool;
    public int access_flags;
    public int this_class;
    public int super_class;
55 public int interfaces_count;
    public int[] interfaces;
    public int fields_count;
    public field_info[] fields;
    public int methods_count;
60 public method_info[] methods;
    public int attributes_count;
    public attribute_info[] attributes;

    /** Constructor. Takes in a byte stream representation and transforms
     * each of the attributes in the ClassFile into objects to allow for
     * easier manipulation.
65

```

```

    */
    public ClassFile(InputStream ins)
        throws IOException{
        DataInputStream in = (ins instanceof DataInputStream ?
5         (DataInputStream) ins : new DataInputStream(ins));
        magic = in.readInt();
        minor_version = in.readChar();
        major_version = in.readChar();
        constant_pool_count = in.readChar();
10        constant_pool = new cp_info[constant_pool_count];
        for (int i=1; i<constant_pool_count; i++){
            in.mark(1);
            int s = in.read();
            in.reset();
15            switch (s){
                case 1:
                    constant_pool[i] = new CONSTANT_Utf8_info(this, in);
                    break;
                case 3:
20                    constant_pool[i] = new CONSTANT_Integer_info(this, in);
                    break;
                case 4:
                    constant_pool[i] = new CONSTANT_Float_info(this, in);
                    break;
25                case 5:
                    constant_pool[i] = new CONSTANT_Long_info(this, in);
                    i++;
                    break;
                case 6:
30                    constant_pool[i] = new CONSTANT_Double_info(this, in);
                    i++;
                    break;
                case 7:
                    constant_pool[i] = new CONSTANT_Class_info(this, in);
                    break;
35                case 8:
                    constant_pool[i] = new CONSTANT_String_info(this, in);
                    break;
                case 9:
40                    constant_pool[i] = new CONSTANT_Fieldref_info(this, in);
                    break;
                case 10:
                    constant_pool[i] = new CONSTANT_Methodref_info(this,
45                    in);
                    break;
                case 11:
                    constant_pool[i] =
                        new CONSTANT_InterfaceMethodref_info(this, in);
                    break;
50                case 12:
                    constant_pool[i] = new CONSTANT_NameAndType_info(this,
                    in);
                    break;
                default:
55                    throw new ClassFormatError("Invalid ConstantPoolTag");
            }
        }
        access_flags = in.readChar();
        this_class = in.readChar();
60        super_class = in.readChar();
        interfaces_count = in.readChar();
        interfaces = new int[interfaces_count];
        for (int i=0; i<interfaces_count; i++){
            interfaces[i] = in.readChar();
65        fields_count = in.readChar();
        fields = new field_info[fields_count];
        for (int i=0; i<fields_count; i++) {

```

```

        fields[i] = new field_info(this, in);
    }
    methods_count = in.readChar();
    methods = new method_info[methods_count];
5   for (int i=0; i<methods_count; i++) {
        methods[i] = new method_info(this, in);
    }
    attributes_count = in.readChar();
    attributes = new attribute_info[attributes_count];
10  for (int i=0; i<attributes_count; i++){
        in.mark(2);
        String s = constant_pool[in.readChar()].toString();
        in.reset();
        if (s.equals("SourceFile"))
15         attributes[i] = new SourceFile_attribute(this, in);
        else if (s.equals("Deprecated"))
            attributes[i] = new Deprecated_attribute(this, in);
        else if (s.equals("InnerClasses"))
            attributes[i] = new InnerClasses_attribute(this, in);
20         else
            attributes[i] = new attribute_info.Unknown(this, in);
    }
}

25  /** Serializes the ClassFile object into a byte stream. */
    public void serialize(OutputStream o)
        throws IOException{
        DataOutputStream out = (o instanceof DataOutputStream ?
30         (DataOutputStream) o : new DataOutputStream(o));
        out.writeInt(magic);
        out.writeChar(minor_version);
        out.writeChar(major_version);
        out.writeChar(constant_pool_count);
        for (int i=1; i<constant_pool_count; i++){
35         constant_pool[i].serialize(out);
            if (constant_pool[i] instanceof CONSTANT_Long_info ||
                constant_pool[i] instanceof CONSTANT_Double_info)
                i++;
        }
        out.writeChar(access_flags);
40         out.writeChar(this_class);
        out.writeChar(super_class);
        out.writeChar(interfaces_count);
        for (int i=0; i<interfaces_count; i++)
45         out.writeChar(interfaces[i]);
        out.writeChar(fields_count);
        for (int i=0; i<fields_count; i++)
            fields[i].serialize(out);
        out.writeChar(methods_count);
50         for (int i=0; i<methods_count; i++)
            methods[i].serialize(out);
        out.writeChar(attributes_count);
        for (int i=0; i<attributes_count; i++)
            attributes[i].serialize(out);
55         // Flush the outputstream just to make sure.
        out.flush();
    }
}

60 }

```

**A14. Code\_attribute.java**  
 Convenience class for representing Code\_attribute structures within ClassFiles.

```

import java.util.*;
import java.lang.*;
65 import java.io.*;

```

```

/**
 * The code[] is stored as a 2D array. */
public final class Code_attribute extends attribute_info{

5     public int max_stack;
      public int max_locals;
      public int code_length;
      public byte[][] code;
10     public int exception_table_length;
      public exception_table[] exception_table;
      public int attributes_count;
      public attribute_info[] attributes;

      /** Internal class that handles the exception table. */
15     public final static class exception_table{
          public int start_pc;
          public int end_pc;
          public int handler_pc;
          public int catch_type;
20     }

      /** Constructor called only by method_info. */
      Code_attribute(ClassFile cf, int ani, int al, int ms, int ml, int cl,
                     byte[][] cd, int etl, exception_table[] et, int ac,
                     attribute_info[] a){
25         super(cf);
         attribute_name_index = ani;
         attribute_length = al;
         max_stack = ms;
         max_locals = ml;
         code_length = cl;
         code = cd;
         exception_table_length = etl;
         exception_table = et;
         attributes_count = ac;
         attributes = a;
30     }

      /** Used during input serialization by ClassFile only. */
40     Code_attribute(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        max_stack = in.readChar();
        max_locals = in.readChar();
        code_length = in.readInt();
        code = new byte[code_length][];
45
        int i = 0;
        for (int pos=0; pos<code_length; i++){
50            in.mark(1);
            int s = in.read();
            in.reset();
            switch (s){
                case 16:
55                case 18:
                case 21:
                case 22:
                case 23:
                case 24:
                case 25:
                case 54:
                case 55:
                case 56:
                case 57:
60                case 58:
                case 169:
                case 188:

```

```

5         case 196:
            code[i] = new byte[2];
            break;
        case 17:
        case 19:
        case 20:
        case 132:
        case 153:
10        case 154:
        case 155:
        case 156:
        case 157:
        case 158:
        case 159:
        case 160:
        case 161:
        case 162:
        case 163:
        case 164:
        case 165:
        case 166:
        case 167:
        case 168:
        case 178:
        case 179:
        case 180:
        case 181:
        case 182:
        case 183:
        case 184:
        case 187:
        case 189:
        case 192:
        case 193:
        case 198:
        case 199:
        case 209:
            code[i] = new byte[3];
            break;
40        case 197:
            code[i] = new byte[4];
            break;
        case 185:
        case 200:
        case 201:
            code[i] = new byte[5];
            break;
        case 170:{
50            int pad = 3 - (pos % 4);
            in.mark(pad+13);
            in.skipBytes(pad+5);
            int low = in.readInt();
            code[i] =
                new byte[pad + 13 + ((in.readInt() - low + 1) * 4)];
55            in.reset();
            break;
        }case 171:{
            int pad = 3 - (pos % 4);
            in.mark(pad+9);
            in.skipBytes(pad+5);
            code[i] = new byte[pad + 9 + (in.readInt() * 8)];
60            in.reset();
            break;
        }default:
            code[i] = new byte[1];
65    }
    in.read(code[i], 0, code[i].length);

```

```

        pos += code[i].length;
    }

    // adjust the array to the new size and store the size
byte[][] temp = new byte[i][];
System.arraycopy(code, 0, temp, 0, i);
code = temp;

exception_table_length = in.readChar();
exception_table =
10     new Code_attribute.exception_table(exception_table_length);
for (i=0; i<exception_table_length; i++){
    exception_table[i] = new exception_table();
    exception_table[i].start_pc = in.readChar();
15     exception_table[i].end_pc = in.readChar();
    exception_table[i].handler_pc = in.readChar();
    exception_table[i].catch_type = in.readChar();
}
attributes_count = in.readChar();
attributes = new attribute_info(attributes_count);
20     for (i=0; i<attributes_count; i++){
        in.mark(2);
        String s = cf.constant_pool[in.readChar()].toString();
        in.reset();
25         if (s.equals("LineNumberTable"))
            attributes[i] = new LineNumberTable_attribute(cf, in);
        else if (s.equals("LocalVariableTable"))
            attributes[i] = new LocalVariableTable_attribute(cf, in);
30         else
            attributes[i] = new attribute_info.Unknown(cf, in);
    }
}

/** Used during output serialization by ClassFile only.
35 */
void serialize(DataOutputStream out)
    throws IOException{
    attribute_length = 12 + code_length +
        (exception_table_length * 8);
40     for (int i=0; i<attributes_count; i++){
        attribute_length += attributes[i].attribute_length + 6;
        super.serialize(out);
        out.writeChar(max_stack);
        out.writeChar(max_locals);
45         out.writeInt(code_length);
        for (int i=0, pos=0; pos<code_length; i++){
            out.write(code[i], 0, code[i].length);
            pos += code[i].length;
        }
50         out.writeChar(exception_table_length);
        for (int i=0; i<exception_table_length; i++){
            out.writeChar(exception_table[i].start_pc);
            out.writeChar(exception_table[i].end_pc);
            out.writeChar(exception_table[i].handler_pc);
55             out.writeChar(exception_table[i].catch_type);
        }
        out.writeChar(attributes_count);
        for (int i=0; i<attributes_count; i++)
            attributes[i].serialize(out);
60     }
}
}

```

#### A15. CONSTANT\_Class\_info.java

65 Convenience class for representing CONSTANT\_Class\_info structures within  
ClassFiles.

```

import java.lang.*;
import java.io.*;

/** Class subtype of a constant pool entry. */
5 public final class CONSTANT_Class_info extends cp_info{

    /** The index to the name of this class. */
    public int name_index = 0;

10    /** Convenience constructor.
    */
    public CONSTANT_Class_info(int index) {
        tag = 7;
        name_index = index;
15    }

    /** Used during input serialization by ClassFile only. */
    CONSTANT_Class_info(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        if (tag != 7)
            throw new ClassFormatError();
        name_index = in.readChar();
20    }

    /** Used during output serialization by ClassFile only. */
    void serialize(DataOutputStream out)
        throws IOException{
        out.writeByte(tag);
        out.writeChar(name_index);
30    }

}
35

```

**A16. CONSTANT\_Double\_info.java**  
 Convenience class for representing CONSTANT\_Double\_info structures within  
 ClassFiles.

```

import java.lang.*;
import java.io.*;

/** Double subtype of a constant pool entry. */
40 public final class CONSTANT_Double_info extends cp_info{

    /** The actual value. */
    public double bytes;

    public CONSTANT_Double_info(double d){
        tag = 6;
        bytes = d;
50    }

    /** Used during input serialization by ClassFile only. */
    CONSTANT_Double_info(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        if (tag != 6)
            throw new ClassFormatError();
        bytes = in.readDouble();
55    }

    /** Used during output serialization by ClassFile only. */
    void serialize(DataOutputStream out)
        throws IOException{
        out.writeByte(tag);
60    }

}
65

```

```

        out.writeDouble(bytes);
        long l = Double.doubleToLongBits(bytes);
    }
}
5

```

**A17. CONSTANT\_Fieldref\_info.java**  
 Convenience class for representing CONSTANT\_Fieldref\_info structures within ClassFiles.

```

import java.lang.*;
import java.io.*;

/** Fieldref subtype of a constant pool entry. */
public final class CONSTANT_Fieldref_info extends cp_info{

    15     /** The index to the class that this field is referencing to. */
    public int class_index;

    /** The name and type index this field is referencing to. */
    20     public int name_and_type_index;

    /** Convenience constructor. */
    public CONSTANT_Fieldref_info(int class_index, int name_and_type_index)
    {
        25         tag = 9;
        this.class_index = class_index;
        this.name_and_type_index = name_and_type_index;
    }

    /** Used during input serialization by ClassFile only. */
    30     CONSTANT_Fieldref_info(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        if (tag != 9)
        35         throw new ClassFormatException();
        class_index = in.readChar();
        name_and_type_index = in.readChar();
    }

    /** Used during output serialization by ClassFile only. */
    40     void serialize(DataOutputStream out)
        throws IOException{
        out.writeByte(tag);
        out.writeChar(class_index);
        out.writeChar(name_and_type_index);
    45     }
}

```

**A18. CONSTANT\_Float\_info.java**  
 Convenience class for representing CONSTANT\_Float\_info structures within ClassFiles.

```

import java.lang.*;
import java.io.*;

55 /** Float subtype of a constant pool entry. */
public final class CONSTANT_Float_info extends cp_info{

    /** The actual value. */
    public float bytes;

    60     public CONSTANT_Float_info(float f){
        tag = 4;
    }
}

```

```

        bytes = f;
    }

    /** Used during input serialization by ClassFile only. */
5    CONSTANT_Float_info(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        if (tag != 4)
            throw new ClassFormatError();
10        bytes = in.readFloat();
    }

    /** Used during output serialization by ClassFile only. */
15    public void serialize(DataOutputStream out)
        throws IOException{
        out.writeByte(4);
        out.writeFloat(bytes);
    }
20 }

```

#### A19. CONSTANT\_Integer\_info.java

Convenience class for representing CONSTANT\_Integer\_info structures within ClassFiles.

```

25 import java.lang.*;
import java.io.*;

    /** Integer subtype of a constant pool entry. */
    public final class CONSTANT_Integer_info extends cp_info{
30
        /** The actual value. */
        public int bytes;

35        public CONSTANT_Integer_info(int b) {
            tag = 3;
            bytes = b;
        }

        /** Used during input serialization by ClassFile only. */
40        CONSTANT_Integer_info(ClassFile cf, DataInputStream in)
            throws IOException{
            super(cf, in);
            if (tag != 3)
                throw new ClassFormatError();
45        bytes = in.readInt();
    }

    /** Used during output serialization by ClassFile only. */
50    public void serialize(DataOutputStream out)
        throws IOException{
        out.writeByte(tag);
        out.writeInt(bytes);
    }
55 }

```

#### A20. CONSTANT\_InterfaceMethodref\_info.java

Convenience class for representing CONSTANT\_InterfaceMethodref\_info structures

60 within ClassFiles.

```

import java.lang.*;
import java.io.*;

```

```

5  /** InterfaceMethodref subtype of a constant pool entry.
    */
    public final class CONSTANT_InterfaceMethodref_info extends cp_info{

        /** The index to the class that this field is referencing to. */
        public int class_index;

        /** The name and type index this field is referencing to. */
        public int name_and_type_index;

        public CONSTANT_InterfaceMethodref_info(int class_index,
                                                int name_and_type_index) {
            tag = 11;
            this.class_index = class_index;
            this.name_and_type_index = name_and_type_index;
        }

        /** Used during input serialization by ClassFile only. */
        CONSTANT_InterfaceMethodref_info(ClassFile cf, DataInputStream in)
            throws IOException{
            super(cf, in);
            if (tag != 11)
                throw new ClassFormatException();
            class_index = in.readChar();
            name_and_type_index = in.readChar();
        }

        /** Used during output serialization by ClassFile only. */
        void serialize(DataOutputStream out)
            throws IOException{
            out.writeByte(tag);
            out.writeChar(class_index);
            out.writeChar(name_and_type_index);
        }
    }

```

#### A21. CONSTANT\_Long\_info.java

Convenience class for representing CONSTANT\_Long\_info structures within ClassFiles.

```

45 import java.lang.*;
    import java.io.*;

    /** Long subtype of a constant pool entry. */
    public final class CONSTANT_Long_info extends cp_info{

        /** The actual value. */
        public long bytes;

        public CONSTANT_Long_info(long b){
            tag = 5;
            bytes = b;
        }

        /** Used during input serialization by ClassFile only. */
        CONSTANT_Long_info(ClassFile cf, DataInputStream in)
            throws IOException{
            super(cf, in);
            if (tag != 5)
                throw new ClassFormatException();
            bytes = in.readLong();
        }
    }

```

```

    /** Used during output serialization by ClassFile only. */
    void serialize(DataOutputStream out)
        throws IOException{
        out.writeByte(tag);
        out.writeLong(bytes);
    }
}

```

## A22. CONSTANT\_Methodref\_info.java

Convenience class for representing CONSTANT\_Methodref\_info structures within

ClassFiles.

```

import java.lang.*;
import java.io.*;

/** Methodref subtype of a constant pool entry.
 */
public final class CONSTANT_Methodref_info extends cp_info{

    /** The index to the class that this field is referencing to. */
    public int class_index;

    /** The name and type index this field is referencing to. */
    public int name_and_type_index;

    public CONSTANT_Methodref_info(int class_index, int name_and_type_index)
    {
        tag = 10;
        this.class_index = class_index;
        this.name_and_type_index = name_and_type_index;
    }

    /** Used during input serialization by ClassFile only. */
    CONSTANT_Methodref_info(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        if (tag != 10)
            throw new ClassFormatError();
        class_index = in.readChar();
        name_and_type_index = in.readChar();
    }

    /** Used during output serialization by ClassFile only. */
    void serialize(DataOutputStream out)
        throws IOException{
        out.writeByte(tag);
        out.writeChar(class_index);
        out.writeChar(name_and_type_index);
    }
}

```

## A23. CONSTANT\_NameAndType\_info.java

Convenience class for representing CONSTANT\_NameAndType\_info structures within

ClassFiles.

```

import java.io.*;
import java.lang.*;

/** NameAndType subtype of a constant pool entry.

```

```

*/
public final class CONSTANT_NameAndType_info extends cp_info{
    /** The index to the Utf8 that contains the name. */
    public int name_index;

    /** The index fo the Utf8 that constains the signature. */
    public int descriptor_index;

    public CONSTANT_NameAndType_info(int name_index, int descriptor_index) {
        tag = 12;
        this.name_index = name_index;
        this.descriptor_index = descriptor_index;
    }

    /** Used during input serialization by ClassFile only. */
    CONSTANT_NameAndType_info(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        if (tag != 12)
            throw new ClassFormatException();
        name_index = in.readChar();
        descriptor_index = in.readChar();
    }

    /** Used during output serialization by ClassFile only. */
    void serialize(DataOutputStream out)
        throws IOException{
        out.writeByte(tag);
        out.writeChar(name_index);
        out.writeChar(descriptor_index);
    }
}

```

#### A24. CONSTANT\_String\_info.java

Convenience class for representing CONSTANT\_String\_info structures within ClassFiles.

```

import java.lang.*;
import java.io.*;

/** String subtype of a constant pool entry.
 */
public final class CONSTANT_String_info extends cp_info{

    /** The index to the actual value of the string. */
    public int string_index;

    public CONSTANT_String_info(int value) {
        tag = 8;
        string_index = value;
    }

    /** ONLY TO BE USED BY CLASSFILE! */
    public CONSTANT_String_info(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        if (tag != 8)
            throw new ClassFormatException();
        string_index = in.readChar();
    }

    /** Output serialization, ONLY TO BE USED BY CLASSFILE! */
    public void serialize(DataOutputStream out)

```

```

        throws IOException{
        out.writeByte(tag);
        out.writeChar(string_index);
5    }

    }

A25. CONSTANT_Utf8_info.java
Convenience class for representing CONSTANT_Utf8_info structures within
10 ClassFiles.

import java.io.*;
import java.lang.*;

/** Utf8 subtype of a constant pool entry.
 * We internally represent the Utf8 info byte array
 * as a String.
 */
public final class CONSTANT_Utf8_info extends cp_info{
20    /** Length of the byte array. */
    public int length;

    /** The actual bytes, represented by a String. */
    public String bytes;
25    /** This constructor should be used for the purpose
     * of part creation. It does not set the parent
     * ClassFile reference.
     */
    public CONSTANT_Utf8_info(String s) {
30        tag = 1;
        length = s.length();
        bytes = s;
    }
35    /** Used during input serialization by ClassFile only. */
    public CONSTANT_Utf8_info(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        if (tag != 1)
40            throw new ClassFormatException();
        length = in.readChar();
        byte[] b = new byte[length];
        in.read(b, 0, length);
45        // WARNING: String constructor is deprecated.
        bytes = new String(b, 0, length);
    }
50    /** Used during output serialization by ClassFile only. */
    public void serialize(DataOutputStream out)
        throws IOException{
        out.writeByte(tag);
        out.writeChar(length);
55        // WARNING: Handling of String coversion here might be problematic.
        out.writeBytes(bytes);
    }
60    public String toString(){
        return bytes;
    }
65 }

```

**A26. ConstantValue\_attribute.java**

Convenience class for representing ConstantValue\_attribute structures within

5 ClassFiles.

```

import java.lang.*;
import java.io.*;

10 /** Attribute that allows for initialization of static variables in
    * classes. This attribute will only reside in a field_info struct.
    */

    public final class ConstantValue_attribute extends attribute_info{

15         public int constantvalue_index;

        public ConstantValue_attribute(ClassFile cf, int ani, int al, int cvi){
            super(cf);
            attribute_name_index = ani;
20             attribute_length = al;
            constantvalue_index = cvi;
        }

        public ConstantValue_attribute(ClassFile cf, DataInputStream in)
25             throws IOException{
            super(cf, in);
            constantvalue_index = in.readChar();
        }

        public void serialize(DataOutputStream out)
30             throws IOException{
            attribute_length = 2;
            super.serialize(out);
            out.writeChar(constantvalue_index);
35         }
    }

```

**A27. cp\_info.java**

40 Convenience class for representing cp\_info structures within ClassFiles.

```

import java.lang.*;
import java.io.*;

45 /** Represents the common interface of all constant pool parts
    * that all specific constant pool items must inherit from.
    */
    public abstract class cp_info{

50         /** The type tag that signifies what kind of constant pool
            * item it is */
            public int tag;

            /** Used for serialization of the object back into a bytestream. */
55             abstract void serialize(DataOutputStream out) throws IOException;

            /** Default constructor. Simply does nothing. */
            public cp_info() {}

60             /** Constructor simply takes in the ClassFile as a reference to
                * it's parent
                */
            public cp_info(ClassFile cf) {}

```

```

    /** Used during input serialization by ClassFile only. */
    cp_info(ClassFile cf, DataInputStream in)
        throws IOException{
        tag = in.readUnsignedByte();
5    }

    )

10  A28. Deprecated_attribute.java
    Convenience class for representing Deprecated_attribute structures within ClassFiles.

    import java.lang.*;
    import java.io.*;

15  /** A fix attributed that can be located either in the ClassFile,
     *   field_info or the method_info attribute. Mark deprecated to
     *   indicate that the method, class or field has been superceded.
     */
    public final class Deprecated_attribute extends attribute_info{

20      public Deprecated_attribute(ClassFile cf, int ani, int al){
        super(cf);
        attribute_name_index = ani;
        attribute_length = al;

25      }

    /** Used during input serialization by ClassFile only. */
    Deprecated_attribute(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);

30      }

    }

35  A29. Exceptions_attribute.java
    Convenience class for representing Exceptions_attribute structures within ClassFiles.

    import java.lang.*;
    import java.io.*;

40  /** This is the struct where the exceptions table are located.
     *   <br><br>
     *   This attribute can only appear once in a method_info struct.
     */
    public final class Exceptions_attribute extends attribute_info{

45      public int number_of_exceptions;
      public int[] exception_index_table;

    public Exceptions_attribute(ClassFile cf, int ani, int al, int noe,
                                int[] eit){

50      super(cf);
      attribute_name_index = ani;
      attribute_length = al;
      number_of_exceptions = noe;
      exception_index_table = eit;

55      }

    /** Used during input serialization by ClassFile only. */
    Exceptions_attribute(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        number_of_exceptions = in.readChar();
        exception_index_table = new int[number_of_exceptions];
        for (int i=0; i<number_of_exceptions; i++)

65      }

```

```

        exception_index_table[i] = in.readChar();
    }

    /** Used during output serialization by ClassFile only. */
5   public void serialize(DataOutputStream out)
        throws IOException{
        attribute_length = 2 + (number_of_exceptions*2);
        super.serialize(out);
        out.writeChar(number_of_exceptions);
10        for (int i=0; i<number_of_exceptions; i++){
            out.writeChar(exception_index_table[i]);
        }
    }
}

15
A30. field_info.java
Convenience class for representing field_info structures within ClassFiles.

import java.lang.*;
import java.io.*;

20 /** Represents the field_info structure as specified in the JVM
    specification.
    */
    public final class field_info{

25        public int access_flags;
        public int name_index;
        public int descriptor_index;
        public int attributes_count;
30        public attribute_info[] attributes;

        /** Convenience constructor. */
        public field_info(ClassFile cf, int flags, int ni, int di){
            access_flags = flags;
35            name_index = ni;
            descriptor_index = di;
            attributes_count = 0;
            attributes = new attribute_info[0];
        }

40        /** Constructor called only during the serialization process.
         * <br><br>
         * This is intentionally left as package protected as we
         * should not normally call this constructor directly.
45         * <br><br>
         * Warning: the handling of len is not correct (after String s=...)
         */
        field_info(ClassFile cf, DataInputStream in)
            throws IOException{
50            access_flags = in.readChar();
            name_index = in.readChar();
            descriptor_index = in.readChar();
            attributes_count = in.readChar();
            attributes = new attribute_info[attributes_count];
55            for (int i=0; i<attributes_count; i++){
                in.mark(2);
                String s = cf.constant_pool[in.readChar()].toString();
                in.reset();
                if (s.equals("ConstantValue"))
                    attributes[i] = new ConstantValue_attribute(cf, in);
60                else if (s.equals("Synthetic"))
                    attributes[i] = new Synthetic_attribute(cf, in);
                else if (s.equals("Deprecated"))
                    attributes[i] = new Deprecated_attribute(cf, in);
65                else
                    attributes[i] = new attribute_info.Unknown(cf, in);
            }
        }
    }
}

```

```

    )
    /** To serialize the contents into the output format.
    */
    public void serialize(DataOutputStream out)
        throws IOException{
        out.writeChar(access_flags);
        out.writeChar(name_index);
        out.writeChar(descriptor_index);
        out.writeChar(attributes_count);
        for (int i=0; i<attributes_count; i++)
            attributes[i].serialize(out);
    }
}

A31. InnerClasses_attribute.java
Convenience class for representing InnerClasses_attribute structures within ClassFiles.

import java.lang.*;
import java.io.*;

/** A variable length structure that contains information about an
 * inner class of this class.
 */
public final class InnerClasses_attribute extends attribute_info{

    public int number_of_classes;
    public classes[] classes;

    public final static class classes{
        int inner_class_info_index;
        int outer_class_info_index;
        int inner_name_index;
        int inner_class_access_flags;
    }

    public InnerClasses_attribute(ClassFile cf, int ani, int al,
                                int noc, classes[] c){
        super(cf);
        attribute_name_index = ani;
        attribute_length = al;
        number_of_classes = noc;
        classes = c;
    }

    /** Used during input serialization by ClassFile only. */
    InnerClasses_attribute(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        number_of_classes = in.readChar();
        classes = new InnerClasses_attribute.classes[number_of_classes];
        for (int i=0; i<number_of_classes; i++){
            classes[i] = new classes();
            classes[i].inner_class_info_index = in.readChar();
            classes[i].outer_class_info_index = in.readChar();
            classes[i].inner_name_index = in.readChar();
            classes[i].inner_class_access_flags = in.readChar();
        }
    }

    /** Used during output serialization by ClassFile only. */
    public void serialize(DataOutputStream out)
        throws IOException{
        attribute_length = 2 + (number_of_classes * 8);
        super.serialize(out);
    }
}

```

```

        out.writeChar(number_of_classes);
        for (int i=0; i<number_of_classes; i++){
            out.writeChar(classes[i].inner_class_info_index);
            out.writeChar(classes[i].outer_class_info_index);
            out.writeChar(classes[i].inner_name_index);
            out.writeChar(classes[i].inner_class_access_flags);
        }
    }
}

A32. LineNumberTable_attribute.java
Convenience class for representing LineNumberTable_attribute structures within
ClassFiles.

import java.lang.*;
import java.io.*;

/** Determines which line of the binary code relates to the
 * corresponding source code.
 */
public final class LineNumberTable_attribute extends attribute_info{

    public int line_number_table_length;
    public line_number_table[] line_number_table;

    public final static class line_number_table{
        int start_pc;
        int line_number;
    }

    public LineNumberTable_attribute(ClassFile cf, int ani, int al, int
    lnt1, line_number_table[] lnt){
        super(cf);
        attribute_name_index = ani;
        attribute_length = al;
        line_number_table_length = lnt1;
        line_number_table = lnt;
    }

    /** Used during input serialization by ClassFile only. */
    LineNumberTable_attribute(ClassFile cf, DataInputStream in)
    throws IOException{
        super(cf, in);
        line_number_table_length = in.readChar();
        line_number_table = new
        LineNumberTable_attribute.line_number_table[line_number_table_length];
        for (int i=0; i<line_number_table_length; i++){
            line_number_table[i] = new line_number_table();
            line_number_table[i].start_pc = in.readChar();
            line_number_table[i].line_number = in.readChar();
        }
    }

    /** Used during output serialization by ClassFile only. */
    void serialize(DataOutputStream out)
    throws IOException{
        attribute_length = 2 + (line_number_table_length * 4);
        super.serialize(out);
        out.writeChar(line_number_table_length);
        for (int i=0; i<line_number_table_length; i++){
            out.writeChar(line_number_table[i].start_pc);
            out.writeChar(line_number_table[i].line_number);
        }
    }
}

```

```

    }

```

### A33. LocalVariableTable\_attribute.java

- 5 Convenience class for representing LocalVariableTable\_attribute structures within ClassFiles.

```

import java.lang.*;
import java.io.*;

10 /** Used by debugger to find out how the source file line number is linked
    * to the binary code. It has many to one correspondence and is found in
    * the Code_attribute.
    */
15 public final class LocalVariableTable_attribute extends attribute_info{

    public int local_variable_table_length;
    public local_variable_table[] local_variable_table;

20     public final static class local_variable_table{
        int start_pc;
        int length;
        int name_index;
        int descriptor_index;
        int index;

25     }

    public LocalVariableTable_attribute(ClassFile cf, int ani, int al,
                                       int lvtl, local_variable_table[]
30     lvt){
        super(cf);
        attribute_name_index = ani;
        attribute_length = al;
        local_variable_table_length = lvtl;
        local_variable_table = lvt;

35     }

    /** Used during input serialization by ClassFile only. */
    LocalVariableTable_attribute(ClassFile cf, DataInputStream in)
        throws IOException{
40         super(cf, in);
        local_variable_table_length = in.readChar();
        local_variable_table = new
LocalVariableTable_attribute.local_variable_table[local_variable_table_length
45     ];
        for (int i=0; i<local_variable_table_length; i++){
            local_variable_table[i] = new local_variable_table();
            local_variable_table[i].start_pc = in.readChar();
            local_variable_table[i].length = in.readChar();
            local_variable_table[i].name_index = in.readChar();
50             local_variable_table[i].descriptor_index = in.readChar();
            local_variable_table[i].index = in.readChar();
        }

    }

55     /** Used during output serialization by ClassFile only. */
    void serialize(DataOutputStream out)
        throws IOException{
        attribute_length = 2 + (local_variable_table_length * 10);
60         super.serialize(out);
        out.writeChar(local_variable_table_length);
        for (int i=0; i<local_variable_table_length; i++){
            out.writeChar(local_variable_table[i].start_pc);
            out.writeChar(local_variable_table[i].length);
            out.writeChar(local_variable_table[i].name_index);
65             out.writeChar(local_variable_table[i].descriptor_index);
        }
    }

```

```

        out.writeChar(local_variable_table[i].index);
    }
}

5  }

A34. method_info.java
Convenience class for representing method_info structures within ClassFiles.

import java.lang.*;
import java.io.*;

/** This follows the method_info in the JVM specification.
 */
15 public final class method_info {

    public int access_flags;
    public int name_index;
    public int descriptor_index;
    public int attributes_count;
20 public attribute_info[] attributes;

    /** Constructor. Creates a method_info, initializes it with
     * the flags set, and the name and descriptor indexes given.
     * A new uninitialized code attribute is also created, and stored
25 * in the <i>code</i> variable.*/
    public method_info(ClassFile cf, int flags, int ni, int di,
        int ac, attribute_info[] a) {
        access_flags = flags;
        name_index = ni;
30 descriptor_index = di;
        attributes_count = ac;
        attributes = a;
    }

35 /** This method creates a method_info from the current pointer in the
     * data stream. Only called by during the serialization of a complete
     * ClassFile from a bytestream, not normally invoked directly.
     */
    method_info(ClassFile cf, DataInputStream in)
40 throws IOException{
        access_flags = in.readChar();
        name_index = in.readChar();
        descriptor_index = in.readChar();
        attributes_count = in.readChar();
45 attributes = new attribute_info(attributes_count);
        for (int i=0; i<attributes_count; i++){
            in.mark(2);
            String s = cf.constant_pool[in.readChar()].toString();
            in.reset();
50 if (s.equals("Code"))
                attributes[i] = new Code_attribute(cf, in);
            else if (s.equals("Exceptions"))
                attributes[i] = new Exceptions_attribute(cf, in);
            else if (s.equals("Synthetic"))
                attributes[i] = new Synthetic_attribute(cf, in);
55 else if (s.equals("Deprecated"))
                attributes[i] = new Deprecated_attribute(cf, in);
            else
                attributes[i] = new attribute_info.Unknown(cf, in);
60 }
    }

    /** Output serialization of the method_info to a byte array.
     * Not normally invoked directly.
65 */
    public void serialize(DataOutputStream out)

```

```

        throws IOException{
            out.writeChar(access_flags);
            out.writeChar(name_index);
            out.writeChar(descriptor_index);
            out.writeChar(attributes_count);
            for (int i=0; i<attributes_count; i++)
                attributes[i].serialize(out);
        }
    }
}

```

### A35. SourceFile\_attribute.java

Convenience class for representing SourceFile\_attribute structures within ClassFiles.

```

import java.lang.*;
import java.io.*;

/** A SourceFile attribute is an optional fixed length attribute in
 * the attributes table. Only located in the ClassFile struct only
 * once.
 */
public final class SourceFile_attribute extends attribute_info{

    public int sourcefile_index;

    public SourceFile_attribute(ClassFile cf, int ani, int al, int sfi){
        super(cf);
        attribute_name_index = ani;
        attribute_length = al;
        sourcefile_index = sfi;
    }

    /** Used during input serialization by ClassFile only. */
    SourceFile_attribute(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        sourcefile_index = in.readChar();
    }

    /** Used during output serialization by ClassFile only. */
    void serialize(DataOutputStream out)
        throws IOException{
        attribute_length = 2;
        super.serialize(out);
        out.writeChar(sourcefile_index);
    }
}

```

### A36. Synthetic\_attribute.java

Convenience class for representing Synthetic\_attribute structures within ClassFiles.

```

import java.lang.*;
import java.io.*;

/** A synthetic attribute indicates that this class does not have
 * a generated code source. It is likely to imply that the code
 * is generated by machine means rather than coded directly. This
 * attribute can appear in the classfile, method_info or field_info.
 * It is fixed length.
 */
public final class Synthetic_attribute extends attribute_info{

    public Synthetic_attribute(ClassFile cf, int ani, int al){
        super(cf);
    }
}

```

```
        attribute_name_index = ani;  
        attribute_length = al;  
    }  
5    /** Used during output serialization by ClassFile only. */  
    Synthetic_attribute(ClassFile cf, DataInputStream in)  
        throws IOException(  
            super(cf, in);  
    }  
10 }
```

**ANNEXURE B**

Annexure B1 is a before-modification excerpt of the disassembled compiled form of the <clinit> method of the example.java application of Annexure B9.

- 5 Annexure B2 is an after-modification form of Annexure B1, modified by InitLoader.java of Annexure B10 in accordance with the steps of Fig. 20. Annexure B3 is a before-modification excerpt of the disassembled compiled form of the <init> method of the example.java application of Annexure B9. Annexure B4 is an after-modification form of Annexure B3, modified by InitLoader.java of Annexure B10 in accordance with the steps of Fig. 21. Annexure B5 is an alternative after-modification form of Annexure B1, modified by InitLoader.java of Annexure B10 in accordance with the steps of Fig. 20. And Annexure B6 is an alternative after-modification form of Annexure B3, modified by InitLoader.java of Annexure B10 in accordance with the steps of Fig. 21. The modifications are highlighted in **bold**.

15

**B1**

Method &lt;clinit&gt;

0 new #2 &lt;Class example&gt;

3 dup

20

4 invokespecial #3 &lt;Method example()&gt;

7 putstatic #4 &lt;Field example currentExample&gt;

10 return

**B2**

Method &lt;clinit&gt;

25

0 invokestatic #3 &lt;Method boolean isAlreadyLoaded()&gt;

3 ifeq 7

6 return

7 new #5 &lt;Class example&gt;

10 dup

30

11 invokespecial #6 &lt;Method example()&gt;

14 putstatic #7 &lt;Field example example&gt;

17 return

**B3**

Method &lt;init&gt;

35

0 aload\_0

1 invokespecial #1 &lt;Method java.lang.Object()&gt;

4 aload\_0

5 invokestatic #2 &lt;Method long currentTimeMillis()&gt;

8 putfield #3 &lt;Field long timestamp&gt;

40

11 return

**B4**

Method &lt;init&gt;

0 aload\_0

```

1 invokespecial #1 <Method java.lang.Object>
4 invokestatic #2 <Method boolean isAlreadyLoaded()>
7 ifeq 11
10 return
5 11 aload_0
12 invokestatic #4 <Method long currentTimeMillis()>
15 putfield #5 <Field long timestamp>
18 return
B5
10 Method <clinit>
    0 ldc #2 <String "example">
    2 invokestatic #3 <Method boolean isAlreadyLoaded(java.lang.String)>
    5 ifeq 9
    8 return
15 9 new #5 <Class example>
    12 dup
    13 invokespecial #6 <Method example()>
    16 putstatic #7 <Field example currentExample>
    19 return
20 B6
    Method <init>
        0 aload_0
        1 invokespecial #1 <Method java.lang.Object>
        4 aload_0
25 5 invokestatic #2 <Method boolean isAlreadyLoaded(java.lang.Object)>
    8 ifeq 12
    11 return
    12 aload_0
    13 invokestatic #4 <Method long currentTimeMillis()>
30 16 putfield #5 <Field long timestamp>
    19 return

```

#### ANNEXURE B7

35 This excerpt is the source-code of InitClient.java, which corresponds to steps 171, 172, 173, 174, 175, and 176 of Fig. 17 and steps 181 and 182 of Fig. 18, and which queries an "initialisation server", such as a machine X of Fig. 15, executing InitServer.java of Annexure B8, in order to determine the initialisation status of the

40 relevant class or object seeking to be initialized.

```

import java.lang.*;
import java.util.*;
import java.net.*;
45 import java.io.*;

```

```

public class InitClient{

    /** Protocol specific values. */
5     public final static int CLOSE = -1;
    public final static int NACK = 0;
    public final static int ACK = 1;
    public final static int INITIALIZE_CLASS = 10;
    public final static int INITIALIZE_OBJECT = 20;
10
    /** InitServer network values. */
    public final static String serverAddress =
        System.getProperty("InitServer_network_address");
    public final static int serverPort =
15        Integer.parseInt(System.getProperty("InitServer_network_port"));

    /** Table of global ID's for local objects. (hashCode-to-globalID
        mappings) */
    public final static Hashtable hashCodeToGlobalID = new Hashtable();
20

    /** Called when a object is being initialized. */
    public static boolean isAlreadyLoaded(Object o){

        // First of all, we need to resolve the globalID
25        // for object 'o'. To do this we use the hashCodeToGlobalID
        // table.
        int globalID = ((Integer) hashCodeToGlobalID.get(o)).intValue();

        try{
30

            // Next, we want to connect to the InitServer, which will inform us
            // of the initialization status of this object.
            Socket socket = new Socket(serverAddress, serverPort);
            DataOutputStream out =
35                new DataOutputStream(socket.getOutputStream());
            DataInputStream in =
                new DataInputStream(socket.getInputStream());

            // Ok, now send the serialized request to the InitServer.
40            out.writeInt(INITIALIZE_OBJECT);
            out.writeInt(globalID);
            out.flush();

            // Now wait for the reply.
45            int status = in.readInt();    // This is a blocking call. So we

```

```

// will wait until the remote side
// sends something.

    if (status == NACK){
        throw new AssertionError(
            "Negative acknowledgement. Request failed.");
    } else if (status != ACK){
        throw new AssertionError("Unknown acknowledgement: "
            + status + ". Request failed.");
    }

    // Next, read in a 32bit argument which is the count of previous
    // initializations.
    int count = in.readInt();

    // If the count is equal to 0, then this is the first
    // initialization, and hence isAlreadyLoaded should be false.
    // If however, the count is greater than 0, then this is already
    // initialized, and thus isAlreadyLoaded should be true.
    boolean isAlreadyLoaded = (count == 0 ? false : true);

    // Close down the connection.
    out.writeInt(CLOSE);
    out.flush();
    out.close();
    in.close();

    socket.close(); // Make sure to close the socket.

    // Return the value of the isAlreadyLoaded variable.
    return isAlreadyLoaded;

} catch (IOException e){
    throw new AssertionError("Exception: " + e.toString());
}

}

/** Called when a class is being initialized. */
public static boolean isAlreadyLoaded(String name){

    try{

        // First of all, we want to connect to the InitServer, which will
        // inform us of the initialization status of this class.
        Socket socket = new Socket(serverAddress, serverPort);

```

```

    DataOutputStream out =
        new DataOutputStream(socket.getOutputStream());
    DataInputStream in =
        new DataInputStream(socket.getInputStream());

5
    // Ok, now send the serialized request to the InitServer.
    out.writeInt(INITIALIZE_CLASS);
    out.writeInt(name.length());           // A 32bit length argument of
                                           // the String name.
10    out.write(name.getBytes(), 0, name.length()); // The byte-
                                                    // encoded
                                                    // String name.

    out.flush();

15    // Now wait for the reply.
    int status = in.readInt();             // This is a blocking call. So we
                                           // will wait until the remote side
                                           // sends something.

20    if (status == NACK){
        throw new AssertionError(
            "Negative acknowledgement. Request failed.");
    } else if (status != ACK){
        throw new AssertionError("Unknown acknowledgement: "
25        + status + ". Request failed.");
    }

    // Next, read in a 32bit argument which is the count of the
    // previous initializations.
30    int count = in.readInt();

    // If the count is equal to 0, then this is the first
    // initialization, and hence isAlreadyLoaded should be false.
    // If however, the count is greater than 0, then this is already
35    // loaded, and thus isAlreadyLoaded should be true.
    boolean isAlreadyLoaded = (count == 0 ? false : true);

    // Close down the connection.
    out.writeInt(CLOSE);
40    out.flush();
    out.close();
    in.close();

    socket.close();           // Make sure to close the socket.
45

```

```

        // Return the value of the isAlreadyLoaded variable.
        return isAlreadyLoaded;

    } catch (IOException e){
5         throw new AssertionError("Exception: " + e.toString());
    }
}
}

```

10

### ANNEXURE B8

This excerpt is the source-code of InitServer.java, which corresponds to steps 191, 192, 193, 194, 195, and 196 of Fig. 19, and which operates on an "initialization server" such as a machine X of Fig. 15, and receives an 'initialisation status request' for a specified object or class of a plurality of similar equivalent objects or classes on the plurality of machines M1...Mn, from network 53 and sent by a machine executing InitClient.java of Annexure B7, and in response returns the corresponding initialization status of the specified class or object.

```

20 import java.lang.*;
import java.util.*;
import java.net.*;
import java.io.*;

25 public class InitServer implements Runnable{

    /** Protocol specific values */
    public final static int CLOSE = -1;
    public final static int NACK = 0;
30    public final static int ACK = 1;
    public final static int INITIALIZE_CLASS = 10;
    public final static int INITIALIZE_OBJECT= 20;
    /** InitServer network values. */
    public final static int serverPort = 20001;
35    /** Table of initialization records. */
    public final static Hashtable initializations = new Hashtable();

    /** Private input/output objects. */
    private Socket socket = null;
40    private DataOutputStream outputStream;
    private DataInputStream inputStream;
    private String address;

```

```

public static void main(String[] s)
throws Exception{

5      System.out.println("InitServer_network_address="
          + InetAddress.getLocalHost().getHostAddress());
      System.out.println("InitServer_network_port=" + serverPort);

      // Create a serversocket to accept incoming initialization operation
10     // connections.
      ServerSocket serverSocket = new ServerSocket(serverPort);

      while (!Thread.interrupted()){

15         // Block until an incoming initialization operation connection.
         Socket socket = serverSocket.accept();

         // Create a new instance of InitServer to manage this
         // initialization operation connection.
20         new Thread(new InitServer(socket)).start();

      )

  )

25  /** Constructor. Initialize this new InitServer instance with necessary
      resources for operation. */
  public InitServer(Socket s){
      socket = s;
30      try{
          outputStream = new DataOutputStream(s.getOutputStream());
          inputStream = new DataInputStream(s.getInputStream());
          address = s.getInetAddress().getHostAddress();
      }catch (IOException e){
35          throw new AssertionError("Exception: " + e.toString());
      }
  }

  /** Main code body. Decode incoming initialization operation requests and
      execute accordingly. */
40  public void run(){

      try{

          // All commands are implemented as 32bit integers.
45          // Legal commands are listed in the "protocol specific values"

```

```

// fields above.
int command = inputStream.readInt();

// Continue processing commands until a CLOSE operation.
5 while (command != CLOSE){

    if (command == INITIALIZE_CLASS){ // This is an
                                      // INITIALIZE_CLASS
                                      // operation.

10 // Read in a 32bit length field 'l', and a String name for
    // this class of length 'l'.
    int length = inputStream.readInt();
    byte[] b = new byte[length];
15 inputStream.read(b, 0, b.length);
    String className = new String(b, 0, length);

    // Synchronize on the initializations table in order to
    // ensure thread-safety.
20 synchronized (initializations){

    // Locate the previous initializations entry for this
    // class, if any.
    Integer entry = (Integer) initializations.get(className);

25 if (entry == null){ // This is an unknown class so
                      // update the table with a
                      // corresponding entry.

30 initializations.put(className, new Integer(1));

    // Send a positive acknowledgement to InitClient,
    // together with the count of previous initializations
    // of this class - which in this case of an unknown
35 // class must be 0.
    outputStream.writeInt(ACK);
    outputStream.writeInt(0);
    outputStream.flush();

40 }else{ // This is a known class, so update
        // the count of initializations.

        initializations.put(className,
45 new Integer(entry.intValue() + 1));

```

```

        // Send a positive acknowledgement to InitClient,
        // together with the count of previous initializations
        // of this class - which in this case of a known class
        // must be the value of "entry.intValue()".
5      outputStream.writeInt(ACK);
      outputStream.writeInt(entry.intValue());
      outputStream.flush();

    }

10   }

    }else if (command == INITIALIZE_OBJECT){ // This is an
        // INITIALIZE_OBJECT
15        // operation.

        // Read in the globalID of the object to be initialized.
        int globalID = inputStream.readInt();

20        // Synchronize on the initializations table in order to
        // ensure thread-safety.
        synchronized (initializations){

            // Locate the previous initializations entry for this
            // object, if any.
25        Integer entry = (Integer) initializations.get(
            new Integer(globalID));

            if (entry == null){ // This is an unknown object so
30                // update the table with a
                // corresponding entry.

                initializations.put(new Integer(globalID),
                    new Integer(1));

35                // Send a positive acknowledgement to InitClient,
                // together with the count of previous initializations
                // of this object - which in this case of an unknown
                // object must be 0.
                outputStream.writeInt(ACK);
                outputStream.writeInt(0);
                outputStream.flush();

40            }else{ // This is a known object so update the
                    // count of initializations.
45

```

```
        initializations.put(new Integer(globalID),
            new Integer(entry.intValue() + 1));

5          // Send a positive acknowledgement to InitClient,
          // together with the count of previous initializations
          // of this object - which in this case of a known
          // object must be value "entry.intValue()".
          outputStream.writeInt(ACK);
10         outputStream.writeInt(entry.intValue());
          outputStream.flush();

      }

15     )

    }else(          // Unknown command.
        throw new AssertionError{
            "Unknown command. Operation failed.";
20     }

        // Read in the next command.
        command = inputStream.readInt();

25     )

}catch (Exception e){
    throw new AssertionError("Exception: " + e.toString());
}finally{
30     try{
        // Closing down. Cleanup this connection.
        outputStream.flush();
        outputStream.close();
        inputStream.close();
35     socket.close();
    }catch (Throwable t){
        t.printStackTrace();
    }
    // Garbage these references.
40     outputStream = null;
    inputStream = null;
    socket = null;
}

45 )
```

}

ANNEXURE B9

5

This excerpt is the source-code of the example.java application used in before/after modification excerpts B1-B6. This example application has two initialization routines, <clinit> and <init> which are modified in accordance with this invention by InitLoader.java of Annexure B10.

10

```
import java.lang.*;
import java.io.*;
```

15

```
public class example implements Serializable{
```

20

```
    /** Shared static field. */
    public static example currentExample;
```

```
    /** Shared instance field. */
    public long timestamp;
```

```
    /** Static initializer. (clinit) */
    static{
```

25

```
        currentExample = new example();
```

```
    }
```

30

```
    /** Instance initializer (init) */
    public example(){
```

```
        timestamp = System.currentTimeMillis();
```

```
    }
```

35

```
}
```

ANNEXURE B10**InitLoader.java**

40

This excerpt is the source-code of InitLoader.java, which modifies an application program code, such as the example.java application code of Annexure B9, as it is being loaded into a JAVA virtual machine in accordance with steps 161-164 of Fig. 16, and steps 201-204 of Fig. 20, and steps 201, 212, 213, and 204 of Fig. 21.

- 5 InitLoader.java makes use of the convenience classes of Annexures A12 through A36 during the modification of a compiled JAVA classfile.

```

import java.lang.*;
import java.io.*;
10 import java.net.*;

public class InitLoader extends URLClassLoader{

    public InitLoader(URL[] urls){
15         super(urls);
    }

    protected Class findClass(String name)
    throws ClassNotFoundException{

20         ClassFile cf = null;

        try{
            BufferedInputStream in = new
25             BufferedInputStream(findResource(name.replace('.',
                '/').concat(".class")).openStream());

            cf = new ClassFile(in);

30         }catch (Exception e){throw new ClassNotFoundException(e.toString());}

        for (int i=0; i<cf.methods_count; i++){

            // Find the <clinit> method_info struct.
35             String methodName = cf.constant_pool[
                cf.methods[i].name_index].toString();
            if (!methodName.equals("<clinit>")){
                continue;
            }

40             // Now find the Code_attribute for the <clinit> method.
            for (int j=0; j<cf.methods[i].attributes_count; j++){
                if (!(cf.methods[i].attributes[j] instanceof Code_attribute))
                    continue;

45                 Code_attribute ca = (Code_attribute)
                    cf.methods[i].attributes[j];

                    // First, shift the code[] down by 4 instructions.
50                     byte[][] code2 = new byte[ca.code.length+4][];
                    System.arraycopy(ca.code, 0, code2, 4, ca.code.length);
                    ca.code = code2;

                    // Then enlarge the constant_pool by 7 items.
55                     cp_info[] cpi = new cp_info[cf.constant_pool.length+7];
                    System.arraycopy(cf.constant_pool, 0, cpi, 0,
                        cf.constant_pool.length);
                    cf.constant_pool = cpi;
                    cf.constant_pool_count += 7;

60

```

```

starting    // Now add the constant pool items for these instructions,
            // with String.
            CONSTANT_String_info csi = new CONSTANT_String_info(
5            ((CONSTANT_Class_info)cf.constant_pool[cf.this_class]).name_index);
            cf.constant_pool[cf.constant_pool.length-7] = csi;

            // Now add the UTF for class.
            CONSTANT_Utf8_info u1 = new CONSTANT_Utf8_info("InitClient");
            cf.constant_pool[cf.constant_pool.length-6] = u1;

            // Now add the CLASS for the previous UTF.
            CONSTANT_Class_info c1 =
15            new CONSTANT_Class_info(cf.constant_pool.length-6);
            cf.constant_pool[cf.constant_pool.length-5] = c1;

            // Next add the first UTF for NameAndType.
            u1 = new CONSTANT_Utf8_info("isAlreadyLoaded");
            cf.constant_pool[cf.constant_pool.length-4] = u1;

            // Next add the second UTF for NameAndType.
            u1 = new CONSTANT_Utf8_info("(Ljava/lang/String;)Z");
            cf.constant_pool[cf.constant_pool.length-3] = u1;

            // Next add the NameAndType for the previous two UTFs.
            CONSTANT_NameAndType_info n1 = new CONSTANT_NameAndType_info(
25            cf.constant_pool.length-4, cf.constant_pool.length-3);
            cf.constant_pool[cf.constant_pool.length-2] = n1;

            // Next add the Methodref for the previous CLASS and
            NameAndType.
            CONSTANT_Methodref_info m1 = new CONSTANT_Methodref_info(
35            cf.constant_pool.length-5, cf.constant_pool.length-3);
            cf.constant_pool[cf.constant_pool.length-1] = m1;

            // Now with that done, add the instructions into the code,
            starting
            // with LDC.
            ca.code[0] = new byte[3];
            ca.code[0][0] = (byte) 19;
            ca.code[0][1] = (byte) (((cf.constant_pool.length-7) >> 8) &
40            0xff);
            ca.code[0][2] = (byte) ((cf.constant_pool.length-7) & 0xff);

            // Now Add the INVOKESTATIC instruction.
            ca.code[1] = new byte[3];
            ca.code[1][0] = (byte) 184;
            ca.code[1][1] = (byte) (((cf.constant_pool.length-1) >> 8) &
50            0xff);
            ca.code[1][2] = (byte) ((cf.constant_pool.length-1) & 0xff);

            // Next add the IFEQ instruction.
            ca.code[2] = new byte[3];
            ca.code[2][0] = (byte) 153;
            ca.code[2][1] = (byte) ((4 >> 8) & 0xff);
            ca.code[2][2] = (byte) (4 & 0xff);

            // Finally, add the RETURN instruction.
            ca.code[3] = new byte[1];
            ca.code[3][0] = (byte) 177;

            // Lastly, increment the CODE_LENGTH and ATTRIBUTE_LENGTH
            values.
65            ca.code_length += 10;
            ca.attribute_length += 10;

```

```
        }  
    }  
5      try{  
        ByteArrayOutputStream out = new ByteArrayOutputStream();  
        cf.serialize(out);  
10       byte[] b = out.toByteArray();  
        return defineClass(name, b, 0, b.length);  
15     }catch (Exception e){  
        e.printStackTrace();  
        throw new ClassNotFoundException(name);  
    }  
20 }  
}
```

ANNEXURE C

Annexure C1 is a before-modification excerpt of the disassembled compiled form of the finalize() method of the example.java application of Annexure C4.

- 5 Annexure C2 is an after-modification form of Annexure C1, modified by FinalLoader.java of Annexure C7 in accordance with the steps of Fig. 22. Annexure C3 is an alternative after-modification form of Annexure C1, modified by FinalLoader.java of Annexure C7 in accordance with the steps of Fig. 22. The modifications are highlighted in **bold**.

10

## C1. TYPICAL PRIOR ART FINALIZATION FOR A SINGLE MACHINE:

```
Method finalize()
0  getstatic #9 <Field java.io.PrintStream out>
3  ldc #24 <String "Deleted...">
15 5  invokevirtual #16 <Method void println(java.lang.String)>
8  return
```

15

## C2. PREFERRED FINALIZATION FOR MULTIPLE MACHINES

```
Method finalize()
0  invokestatic #3 <Method boolean isLastReference()>
20 3  ifne 7
6  return
7  getstatic #9 <Field java.io.PrintStream out>
10 ldc #24 <String "Deleted...">
12  invokevirtual #16 <Method void println(java.lang.String)>
25 15 return
```

20

25

## C3. PREFERRED FINALIZATION FOR MULTIPLE MACHINES (Alternative)

```
Method finalize()
0  aload_0
30 1  invokestatic #3 <Method boolean isLastReference(java.lang.Object)>
4  ifne 8
7  return
8  getstatic #9 <Field java.io.PrintStream out>
11 ldc #24 <String "Deleted...">
13  invokevirtual #16 <Method void println(java.lang.String)>
35 16 return
```

30

35

Annexure C4

- 40 This excerpt is the source-code of the example.java application used in before/after modification excerpts C1-C3. This example application has a single finalization

routine, the finalize() method, which is modified in accordance with this invention by FinalLoader.java of Annexure C7.

```

5  import java.lang.*;

    public class example{

        /** Finalize method. */
10     protected void finalize() throws Throwable{

            // "Deleted..." is printed out when this object is garbaged.
            System.out.println("Deleted...");

15     }
    }

```

#### Annexure C5

20 This excerpt is the source-code of FinalClient.java, which corresponds to steps 171, 172A, 173A, 174A, 175A, and 176A of Fig. 23, and which queries an "finalization server", such as a machine X of Fig. 15, executing InitServer.java of Annexure C6, in order to determine the finalization status of the relevant class or object seeking to be

25 finalized.

```

import java.lang.*;
import java.util.*;
30 import java.net.*;
import java.io.*;

    public class FinalClient{

35     /** Protocol specific values. */
        public final static int CLOSE = -1;
        public final static int NACK = 0;
        public final static int ACK = 1;
        public final static int FINALIZE_OBJECT = 10;

40     /** FinalServer network values. */
        public final static String serverAddress =
            System.getProperty("FinalServer_network_address");

```

```
public final static int serverPort =
    Integer.parseInt(System.getProperty("FinalServer_network_port"));

/** Table of global ID's for local objects. (hashCode-to-globalID
5   mappings) */
public final static Hashtable hashCodeToGlobalID = new Hashtable();

/** Called when a object is being finalized. */
10 public static boolean isLastReference(Object o){

    // First of all, we need to resolve the globalID for object 'o'.
    // To do this we use the hashCodeToGlobalID table.
    int globalID = ((Integer) hashCodeToGlobalID.get(o)).intValue();

15   try{

        // Next, we want to connect to the FinalServer, which will inform
        // us of the finalization status of this object.
        Socket socket = new Socket(serverAddress, serverPort);
        DataOutputStream out =
            new DataOutputStream(socket.getOutputStream());
        DataInputStream in = new DataInputStream(socket.getInputStream());

        // Ok, now send the serialized request to the FinalServer.
25     out.writeInt(FINALIZE_OBJECT);
        out.writeInt(globalID);
        out.flush();

        // Now wait for the reply.
30     int status = in.readInt();    // This is a blocking call. So we
                                    // will wait until the remote side
                                    // sends something.

        if (status == NACK){
35             throw new AssertionError(
                "Negative acknowledgement. Request failed.");
        }else if (status != ACK){
            throw new AssertionError("Unknown acknowledgement: "
40             + status + ". Request failed.");
        }

        // Next, read in a 32bit argument which is the count of the
        // remaining finalizations
        int count = in.readInt();
45
```

```

    // If the count is equal to 1, then this is the last finalization,
    // and hence isLastReference should be true.
    // If however, the count is greater than 1, then this is not the
    // last finalization, and thus isLastReference should be false.
5   boolean isLastReference = (count == 1 ? true : false);

    // Close down the connection.
    out.writeInt(CLOSE);
    out.flush();
10   out.close();
    in.close();

    socket.close();           // Make sure to close the socket.

15   // Return the value of the isLastReference variable.
    return isLastReference;

} catch (IOException e){
    throw new AssertionError("Exception: " + e.toString());
20   }
}
}

```

### Annexure C6

25 This excerpt is the source-code of FinalServer.java, which corresponds to steps 191A, 192A, 193A, 194A, 195A, 196A, and 197A of Fig. 25, and which operates on an "finalization server" such as a machine X of Fig. 15, and receives an 'clean-up status request' for a specified object or class of a plurality of similar equivalent objects or

30 classes on the plurality of machines M1...Mn, from network 53 and sent by a machine executing FinalClient.java of Annexure C5, and in response returns the corresponding finalization status of the specified class or object.

```

35   import java.lang.*;
    import java.util.*;
    import java.net.*;
    import java.io.*;

40   public class FinalServer implements Runnable{

        /** Protocol specific values */

```

```

    public final static int CLOSE = -1;
    public final static int NACK = 0;
    public final static int ACK = 1;
    public final static int FINALIZE_OBJECT = 10;
5
    /** FinalServer network values. */
    public final static int serverPort = 20001;

    /** Table of finalization records. */
10    public final static Hashtable finalizations = new Hashtable();

    /** Private input/output objects. */
    private Socket socket = null;
    private DataOutputStream outputStream;
15    private DataInputStream inputStream;
    private String address;

    public static void main(String[] s)
    throws Exception{
20
        System.out.println("FinalServer_network_address="
            + InetAddress.getLocalHost().getHostAddress());
        System.out.println("FinalServer_network_port=" + serverPort);

25        // Create a serversocket to accept incoming initialization operation
        // connections.
        ServerSocket serverSocket = new ServerSocket(serverPort);

        while (!Thread.interrupted()){
30
            // Block until an incoming initialization operation connection.
            Socket socket = serverSocket.accept();

            // Create a new instance of InitServer to manage this
            // initialization operation connection.
35            new Thread(new FinalServer(socket)).start();

        }

40    }

    /** Constructor. Initialize this new FinalServer instance with necessary
        resources for operation. */
    public FinalServer(Socket s){
45        socket = s;

```

```

        try{
            outputStream = new DataOutputStream(s.getOutputStream());
            inputStream = new DataInputStream(s.getInputStream());
            address = s.getInetAddress().getHostAddress();
5      }catch (IOException e){
            throw new AssertionError("Exception: " + e.toString());
        }
    }

10  /** Main code body. Decode incoming finalization operation requests and
        execute accordingly. */
    public void run(){

15      try{

            // All commands are implemented as 32bit integers.
            // Legal commands are listed in the "protocol specific values"
            // fields above.
            int command = inputStream.readInt();

20      // Continue processing commands until a CLOSE operation.
            while (command != CLOSE){

                if (command == FINALIZE_OBJECT){           // This is a
                                                            // FINALIZE_OBJECT
                                                            // operation.

                    // Read in the globalID of the object to be finalized.
                    int globalID = inputStream.readInt();

30      // Synchronize on the finalizations table in order to ensure
                    // thread-safety.
                    synchronized (finalizations){

85      // Locate the previous finalizations entry for this
                    // object, if any.
                    Integer entry = (Integer) finalizations.get(
                        new Integer(globalID));

40      if (entry == null){
                            throw new AssertionError("Unknown object.");
                        }else if (entry.intValue() < 1){
                            throw new AssertionError("Invalid count.");
                        }else if (entry.intValue() == 1){ // Count of 1 means
                                                            // this is the last
45

```

```

// reference, hence
// remove from table.

finalizations.remove(new Integer(globalID));
5
// Send a positive acknowledgement to FinalClient,
// together with the count of remaining references -
// which in this case is 1.
outputStream.writeInt(ACK);
10 outputStream.writeInt(1);
outputStream.flush();

}else{ // This is not the last remaining
// reference, as count is greater than 1.
15 // Decrement count by 1.

finalizations.put(new Integer(globalID),
new Integer(entry.intValue() - 1));

20 // Send a positive acknowledgement to FinalClient,
// together with the count of remaining references to
// this object - which in this case of must be value
// "entry.intValue()".
outputStream.writeInt(ACK);
25 outputStream.writeInt(entry.intValue());
outputStream.flush();

}

30 }

}else{ // Unknown command.
throw new AssertionError(
"Unknown command. Operation failed.");
35 }

// Read in the next command.
command = inputStream.readInt();

40 }

}catch (Exception e){
throw new AssertionError("Exception: " + e.toString());
}finally{
45 try{
```

```

        // Closing down. Cleanup this connection.
        outputStream.flush();
        outputStream.close();
        inputStream.close();
5       socket.close();
    } catch (Throwable t) {
        t.printStackTrace();
    }
    // Garbage these references.
10    outputStream = null;
    inputStream = null;
    socket = null;
}
15
}

```

### ANNEXURE C7

#### **FinalLoader.java**

25

This excerpt is the source-code of FinalLoader.java, which modifies an application program code, such as the example.java application code of Annexure C4, as it is being loaded into a JAVA virtual machine in accordance with steps 161A, 162A, 163B, and 164A of Fig. 22. FinalLoader.java makes use of the convenience classes of

30 Annexures A12 through to A36 during the modification of a compiled JAVA classfile.

```

import java.lang.*;
import java.io.*;
import java.net.*;
35 public class FinalLoader extends URLClassLoader{
    public FinalLoader(URL[] urls){
        super(urls);
40    }

    protected Class findClass(String name)
    throws ClassNotFoundException{
45        ClassFile cf = null;

        try{
            BufferedInputStream in =
                new BufferedInputStream(findResource(name.replace('.',

```

```

        '/').concat(".class")).openStream());
        cf = new ClassFile(in);
5      }catch (Exception e){throw new ClassNotFoundException(e.toString());}

        for (int i=0; i<cf.methods_count; i++){
10          // Find the finalize_method_info struct.
          String methodName = cf.constant_pool[
            cf.methods[i].name_index].toString();
          if (!methodName.equals("finalize")){
            continue;
15          }

          // Now find the Code_attribute for the finalize method.
          for (int j=0; j<cf.methods[i].attributes_count; j++){
            if (!(cf.methods[i].attributes[j] instanceof Code_attribute))
              continue;
20          Code_attribute ca = (Code_attribute)
            cf.methods[i].attributes[j];

            // First, shift the code[] down by 4 instructions.
25          byte[][] code2 = new byte[ca.code.length+4][];
          System.arraycopy(ca.code, 0, code2, 4, ca.code.length);
          ca.code = code2;

            // Then enlarge the constant_pool by 6 items.
30          cp_info[] cpi = new cp_info[cf.constant_pool.length+6];
          System.arraycopy(cf.constant_pool, 0, cpi, 0,
            cf.constant_pool.length);
          cf.constant_pool = cpi;
          cf.constant_pool_count += 6;
35          // Now add the UTF for class.
          CONSTANT_Utf8_info u1 = new CONSTANT_Utf8_info("FinalClient");
          cf.constant_pool[cf.constant_pool.length-6] = u1;

40          // Now add the CLASS for the previous UTF.
          CONSTANT_Class_info c1 =
            new CONSTANT_Class_info(cf.constant_pool.length-6);
          cf.constant_pool[cf.constant_pool.length-5] = c1;

45          // Next add the first UTF for NameAndType.
          u1 = new CONSTANT_Utf8_info("isLastReference");
          cf.constant_pool[cf.constant_pool.length-4] = u1;

50          // Next add the second UTF for NameAndType.
          u1 = new CONSTANT_Utf8_info("(Ljava/lang/Object;)Z");
          cf.constant_pool[cf.constant_pool.length-3] = u1;

            // Next add the NameAndType for the previous two UTFs.
55          CONSTANT_NameAndType_info n1 = new CONSTANT_NameAndType_info(
            cf.constant_pool.length-4, cf.constant_pool.length-3);
          cf.constant_pool[cf.constant_pool.length-2] = n1;

            // Next add the Methodref for the previous CLASS and
60          NameAndType.
          CONSTANT_Methodref_info m1 = new CONSTANT_Methodref_info(
            cf.constant_pool.length-5, cf.constant_pool.length-2);
          cf.constant_pool[cf.constant_pool.length-1] = m1;

            // Now with that done, add the instructions into the code,
65          starting
            // with LDC.
            ca.code[0] = new byte[1];

```

```

        ca.code[0][0] = (byte) 42;

        // Now Add the INVOKESTATIC instruction.
        ca.code[1] = new byte[3];
5      ca.code[1][0] = (byte) 184;
        ca.code[1][1] = (byte) (((cf.constant_pool.length-1) >> 8) &
0xff);
        ca.code[1][2] = (byte) ((cf.constant_pool.length-1) & 0xff);

10     // Next add the IFNE instruction.
        ca.code[2] = new byte[3];
        ca.code[2][0] = (byte) 154;
        ca.code[2][1] = (byte) (((4 >> 8) & 0xff));
        ca.code[2][2] = (byte) (4 & 0xff);

15     // Finally, add the RETURN instruction.
        ca.code[3] = new byte[1];
        ca.code[3][0] = (byte) 177;

20     // Lastly, increment the CODE_LENGTH and ATTRIBUTE_LENGTH
    values.
        ca.code_length += 8;
        ca.attribute_length += 8;

25     )
    )

30     try{
        ByteArrayOutputStream out = new ByteArrayOutputStream();
        cf.serialize(out);

35         byte[] b = out.toByteArray();

        return defineClass(name, b, 0, b.length);

    }catch (Exception e){
        e.printStackTrace();
40         throw new ClassNotFoundException(name);
    }
}

45 )

```

### Annexure D1

Annexure D1 is a before-modification excerpt of the disassembled compiled  
50 form of the synchronization operation of example.java of Annexure D3, consisting of  
an starting “monitorenter” instruction and ending “monitorexit” instruction. Annexure  
D2 is an after-modification form of Annexure D1, modified by LockLoader.java of  
Annexure D6 in accordance with the steps of Fig. 26. The modifications are  
highlighted in **bold**.

55

```

Method void run()
0  getstatic #2 <Field java.lang.Object LOCK>
3  dup

```

```

4 astore_1
5 monitorenter
6 getstatic #3 <Field int counter>
9 iconst_1
5 10 iadd
11 putstatic #3 <Field int counter>
14 aload_1
15 monitorexit
16 return
10

```

#### Annexure D2

```

Method void run()
0 getstatic #2 <Field java.lang.Object LOCK>
15 3 dup
4 astore_1
5 dup
6 monitorenter
7 invokestatic #23 <Method void acquireLock(java.lang.Object)>
20 10 getstatic #3 <Field int counter>
13 iconst_1
14 iadd
15 putstatic #3 <Field int counter>
18 aload_1
25 19 dup
20 invokestatic #24 <Method void releaseLock(java.lang.Object)>
23 monitorexit
24 return
30

```

#### Annexure D3

This excerpt is the source-code of the example.java application used in before/after modification excerpts D1 and D2. This example application has a single synchronization operation routine, consisting of a starting "monitorenter" instruction and an ending "monitorexit" instruction, which is modified in accordance with this invention by LockLoader of Annexure D6.

```

40 import java.lang.*;

public class example{

45  /** Shared static field. */
    public final static Object LOCK = new Object();

```

```

    /** Shared static field. */
    public static int counter = 0;

    /** Example method using synchronization. This method serves to
5      illustrate the use of synchronization to implement thread-safe
      modification of a shared memory location by potentially multiple
      threads. */
    public void run(){

10      // First acquire the lock, otherwise any memory writes we do will be
      // prone to race-conditions.
      synchronized (LOCK){

          // Now that we have acquired the lock, we can safely modify memory
15      // in a thread-safe manner.
      counter++;

      }

20  }

}

```

#### Annexure D4

25

This excerpt is the source-code of LockClient.java, which corresponds to steps 171B, 172B, 173B, 174B, and 175B of Fig. 27, and steps 181B, 182B, 183B, 184B, and 185B of Fig. 28, and which queries an "synchronization server", such as a machine X of Fig. 15, executing LockServer.java of Annexure D5, in order to synchronize (i.e. by means of an acquire lock and release a lock request pair) the relevant class or object seeking to be synchronized (i.e. seeking to be "locked").

```

import java.lang.*;
import java.util.*;
35 import java.net.*;
import java.io.*;

public class LockClient{

40  /** Protocol specific values. */
    public final static int CLOSE = -1;

```

```

public final static int NACK = 0;
public final static int ACK = 1;
public final static int ACQUIRE_LOCK = 10;
public final static int RELEASE_LOCK = 20;
5  /** LockServer network values. */
    public final static String serverAddress =
        System.getProperty("LockServer_network_address");
    public final static int serverPort =
10        Integer.parseInt(System.getProperty("LockServer_network_port"));

    /** Table of global ID's for local objects. (hashCode-to-globalID
        mappings) */
    public final static Hashtable hashCodeToGlobalID = new Hashtable();

15    /** Called when an application is to acquire a lock. */
    public static void acquireLock(Object o){

        // First of all, we need to resolve the globalID for object 'o'.
        // To do this we use the hashCodeToGlobalID table.
20        int globalID = ((Integer) hashCodeToGlobalID.get(o)).intValue();

        try{

            // Next, we want to connect to the LockServer, which will grant us
25            // the global lock.
            Socket socket = new Socket(serverAddress, serverPort);
            DataOutputStream out =
                new DataOutputStream(socket.getOutputStream());
            DataInputStream in = new DataInputStream(socket.getInputStream());

30            // Ok, now send the serialized request to the lock server.
            out.writeInt(ACQUIRE_LOCK);
            out.writeInt(globalID);
            out.flush();

35            // Now wait for the reply.
            int status = in.readInt();    // This is a blocking call. So we
                                         // will wait until the remote side
                                         // sends something.

40            if (status == NACK){
                throw new AssertionError(
                    "Negative acknowledgement. Request failed.");
            }else if (status != ACK){
45                throw new AssertionError("Unknown acknowledgement: "

```

```

        + status + ". Request failed.");
    }

    // Close down the connection.
5    out.writeInt(CLOSE);
    out.flush();
    out.close();
    in.close();

10    socket.close();           // Make sure to close the socket.

    // This is a good acknowledgement, thus we can return now because
    // global lock is now acquired.
    return;

15    } catch (IOException e) {
        throw new AssertionError("Exception: " + e.toString());
    }

20    }

    /** Called when an application is to release a lock. */
    public static void releaseLock(Object o) {

25        // First of all, we need to resolve the globalID for object 'o'.
        // To do this we use the hashCodeToGlobalID table.
        int globalID = ((Integer) hashCodeToGlobalID.get(o)).intValue();

        try {

30            // Next, we want to connect to the LockServer, which records us as
            // the owner of the global lock for object 'o'.
            Socket socket = new Socket(serverAddress, serverPort);
            DataOutputStream out =
35                new DataOutputStream(socket.getOutputStream());
            DataInputStream in = new DataInputStream(socket.getInputStream());

            // Ok, now send the serialized request to the lock server.
            out.writeInt(RELEASE_LOCK);
40            out.writeInt(globalID);
            out.flush();

            // Now wait for the reply.
            int status = in.readInt();           // This is a blocking call. So we
45            // will wait until the remote side

```

```

// sends something.

    if (status == NACK){
        throw new AssertionError(
5         "Negative acknowledgement. Request failed.");
    }else if (status != ACK){
        throw new AssertionError("Unknown acknowledgement: "
            + status + ". Request failed.");
    }

10
    // Close down the connection.
    out.writeInt(CLOSE);
    out.flush();
    out.close();
15    in.close();

    socket.close();        // Make sure to close the socket.

    // This is a good acknowledgement, return because global lock is
    // now released.
20    return;

    }catch (IOException e){
        throw new AssertionError("Exception: " + e.toString());
25    }

    )

30 }

```

#### Annexure D5

35 This excerpt is the source-code of LockServer.java, which corresponds to steps 191B, 192B, 193B, 194B, 195B, 196B, 197B, and 198B of Fig. 29, and steps 201, 202, 203, 204, 205, 206, 206, 207, and 208 of Fig. 30, and which operates on a "synchronization server" such as a machine X of Fig. 15, and receives an 'acquire lock request' or a 'release lock request' for a specified object or class of a plurality of similar equivalent objects or classes on the plurality of machines M1...Mn, from network 53 and sent by
40 a machine executing LockClient.java of Annexure D4, and in response returns the

corresponding confirmation of ownership of an 'acquire lock request', or optionally confirmation of a 'release lock request', of the specified class or object.

```
5  import java.lang.*;
    import java.util.*;
    import java.net.*;
    import java.io.*;

10  public class LockServer implements Runnable{

    /** Protocol specific values */
    public final static int CLOSE = -1;
    public final static int NACK = 0;
15  public final static int ACK = 1;
    public final static int ACQUIRE_LOCK = 10;
    public final static int RELEASE_LOCK = 20;

    /** LockServer network values. */
20  public final static int serverPort = 20001;

    /** Table of lock records. */
    public final static Hashtable locks = new Hashtable();

25  /** Linked list of waiting LockManager objects. */
    public LockServer next = null;

    /** Address of remote LockClient. */
30  public final String address;

    /** Private input/output objects. */
    private Socket socket = null;
    private DataOutputStream outputStream;
    private DataInputStream inputStream;

35  public static void main(String[] s)
    throws Exception{

    System.out.println("LockServer_network_address="
40      + InetAddress.getLocalHost().getHostAddress());
    System.out.println("LockServer_network_port=" + serverPort);

    // Create a serversocket to accept incoming lock operation
```

```

// connections.
ServerSocket serverSocket = new ServerSocket(serverPort);

while (!Thread.interrupted()){
5
    // Block until an incoming lock operation connection.
    Socket socket = serverSocket.accept();

    // Create a new instance of LockServer to manage this lock
10    // operation connection.
    new Thread(new LockServer(socket)).start();

}

15 }

/** Constructor. Initialise this new LockServer instance with necessary
    resources for operation. */
public LockServer(Socket s){
20    socket = s;
    try{
        outputStream = new DataOutputStream(s.getOutputStream());
        inputStream = new DataInputStream(s.getInputStream());
        address = s.getInetAddress().getHostAddress();
25    }catch (IOException e){
        throw new AssertionError("Exception: " + e.toString());
    }
}

30 /** Main code body. Decode incoming lock operation requests and execute
    accordingly. */
public void run(){
    try{
35
        // All commands are implemented as 32bit integers.
        // Legal commands are listed in the "protocol specific values"
        // fields above.
        int command = inputStream.readInt();

40
        // Continue processing commands until a CLOSE operation.
        while (command != CLOSE){

            if (command == ACQUIRE_LOCK){
45
                // This is an
                // ACQUIRE_LOCK

```